Using Roles with JAWIRO

Yunus Emre Selcuk and Nadia Erdogan

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 introduces a role model named JAWIRO, which enhances Java with role support. JAWIRO implements features expected of roles and adds extended features, without hampering the performance of method calls. As the result, JAWIRO provides a better and feasible means to model dynamically evolving systems.

Introduction

Object oriented programming (OOP) requires that the relationship between an object and its respective class is persistent, static and exclusive. This property of OOP makes it efficient at modeling real world objects that can be divided into distinct classes where objects never change their classes. However, the objects within the real world can display dynamic behavior as they constantly change and evolve by gaining or loosing some abilities and responsibilities. An ability or responsibility of a real world object can be called as a role. In such cases, OOP requires programmers to define classes that determine the behavior of each separate role in the modeled system. After these classes are built, one can use multiple inheritance to create combination classes which represent the roles that can be acquired in parallel. Alternatively, one can somehow glue either the separate classes or the instances of these classes. All of these choices described above are tedious, if possible. For example:

- Some OOP languages such as Java do not support multiple inheritance.
- The combination classes grow exponentially every time a new role is introduced.
- Gluing entities in class level is restrictive and can require modifications to the chosen programming language.

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 (Ungar, 1987), dynamic reclassification (Drossopoulou, 2002), subject oriented programming (Wong, 1997), design patterns (Fowler), etc. A more detailed review of the problem and proposed approaches for modeling dynamically evolving entities can be found in (Selcuk, 2003).

This paper presents a role model implementation, JAWIRO, which enhances Java with role support for better modeling of dynamically evolving real world systems. JAWIRO implements all features expected of roles and adds extended features, with no or small overhead on the performance of the application.

Roles 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. From the modeling perspective, *roles* are different types of behavior that different types of entities can perform. Kristensen (Kristensen, 1996) 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.

Specialization at the instance level is a better approach than specialization at the class level when modeling evolving entities. In this case, an entity is represented by multiple objects, each executing a different role that the real-world entity is required to perform. When class level specialization is used, all instances of that particular class are expected to have the same set of roles. On the other hand, different entities of the same type can have different set of roles in instance level specialization. Such specialization of an object at the instance level by acquiring new roles or loosing some roles is called *object level inheritance*. *Role based programming* (RBP) is a paradigm which utilizes roles in this manner and a *role model* specifies a style of designing and implementing roles. Role models provide a mechanism for object level inheritance while preserving the fact that multiple objects are used to model a real world entity.

RBP extends the concepts of OOP naturally and elegantly. While the class level inheritance elegantly models the *IsA* relation, object level inheritance successfully models the *IsPartOf* relation (Zendler, 1998). An entity is modeled with multiple role objects in RBP; therefore each role object is a part of the real world object. The role objects can access each other by the means of the role model, i.e. role switching. As both types of relationship are required when modeling of real world systems, both types of inheritance should coexist in an

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

object-oriented environment. Therefore, many role models are implemented by extending an object-oriented language of choice. Some examples for such works are INADA extending C++ (Aritsugi, 2000), DEC-JAVA extending Java (Bettini, 2003) and the works of Gottlob et al. extending Smalltalk (Gottlob, 1996).

An Advanced Role Model for Java: Jawiro

The aim of this work is to implement a powerful role model which support all features expected of roles without introducing any restrictions or a significant overhead on performance. Our role model is named after its purpose: Extending Java 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.

Role Model of JAWIRO

JAWIRO role model uses a tree representation for modeling relational hierarchies of roles. A hierarchical representation enables better modeling of role ownership relations, while allowing for an elegant and robust implementation of roles' characteristics.

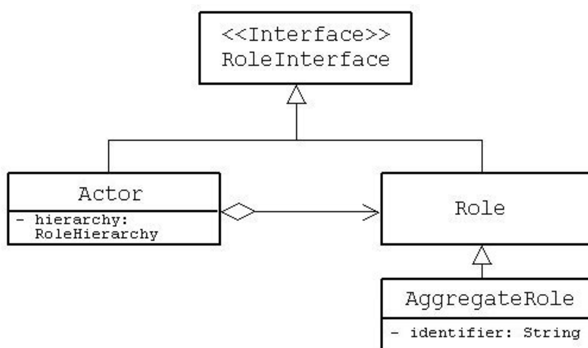

Figure 1: The UML schema of JAWIRO API

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 models the role objects. The `Actor` and `Role` classes implement the `RoleInterface` as these two classes share some characteristics of roles. The *aggregate roles*, which an owner can play multiple instances of this type, 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. More details on the JAWIRO API can be found in (Selcuk, 2004).

Features of Roles

Definition of the basic features of roles varies slightly among different researchers such as (Kristensen, 1996) and (Schrefl, 2004). According to our experience, 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 distinguished from each other with an *identifier*.
- Different roles are allowed to have member variables and methods with same names without conflicts.

These basic features do not cover all that can be done with roles. JAWIRO implements the following extended features of roles as well:

- Roles can be suspended and then resumed.
- A role can be transferred to another owner without dropping its sub roles, e.g. the child roles of this particular role object in the tree shaped role hierarchy.
- 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.

Using JAWIRO

This section illustrates how the roles are used with JAWIRO in order to demonstrate the capabilities of our role model. The basic features of roles are covered in the first subsection and the extended features are covered in separate subsections.

Using the Basic Features of Roles

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 (Gottlob, 1996) and the second hierarchy will be used in the discussion of an extended feature in the next subsection.

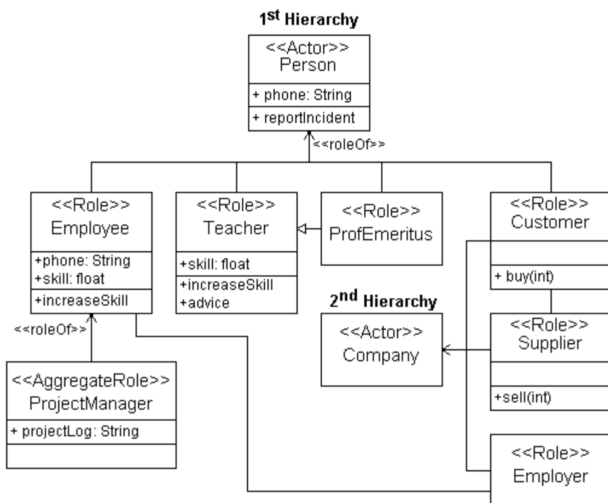

Figure 2: A sample role model containing two intersected hierarchies. A stereotype shows a superclass from the JAWIRO API in Figure 1.

The first six lines of the code in Figure 3 declare the objects to be used. A `Person` instance and one of its roles are created in lines 7 and 8, respectively. Then this person gains the newly created `Teacher` role in line 9. Line 10 demonstrates the run-time role checking feature. The fully qualified class name of the role to be searched is required for the `RoleInterface.canSwitch` method. If this check is successful, and it really is in this example, the role switching is performed and the switched role is executed as shown in line 11. Note that a type casting is required before role execution as Java is a strongly typed language. It is a wise move to check for the role existence first and then performing the role switching. This does not introduce a performance penalty as JAWIRO remembers the last found role with the `canSwitch` command and returns directly this particular role if an `as` command requesting this role is executed. Another way of role checking and execution is given in the next subsection.

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. Lines 13 and 14 of Figure 3 give such an example. After the person looses the teacher role in line 12, he is given a `ProfEmeritus` role so that he can still teach.

The association between the `Employer` and `Employee` classes of Figure 2 is demonstrated in lines 15-20 of Figure 3. The root of the second hierarchy is created and gained an employer role in the same fashion as a person gains a teacher role in lines 15-17. A second person, Tom, is created in line 18, its employee role associated with the employer instance is created in line 19 and Tom becomes an employee in the company in line 20.

```
01: Company coMTCX, coBML; Employer erMTCX;
02: Supplier suBML; Customer cuMTCX, cuTom;
03: Person peTom, peGordon;
04: Teacher teGordon; ProfEmeritus prGordon;
05: ProjectManager pm1, pm2;
06: Employee eeTom, eeGordon;
07: peGordon = new Person("Gordon Freeman",
    "637-252"); //Name and home phone
08: teGordon = new Teacher("Physics");
09: peGordon.addRole(teGordon); //Gordon
    becomes a physics teacher.
10: if( peGordon.canSwitch (
    "examples.Teacher" ) ) { //Run-time role
    checking, 1st way.
11:   ((Teacher)peGordon.as(
      "examples.Teacher")).advice(); }
      //Role switching: Person->Teacher.
12: teGordon.resign(); //Gordon retires,...
13: prGordon = new ProfEmeritus(teGordon);
14: peGordon.addRole(prGordon); //... but
    becomes a Professor Emeritus.
15: coMTCX = new Company("Metacortex");
16: erMTCX = new Employer();
17: coMTCX.addRole(erMTCX); //Metacortex is
    ready to enlist.
18: peTom = new Person("Thomas Anderson",
    "843-663");
19: eeTom = new Employee(erMTCX,"628-749");
20: peTom.addRole(eeTom); //Tom is enlisted.
21: pm1 = new ProjectManager("Virtual
    Reality","VR");
22: eeTom.addRole(pm1);  //Tom  leads  the  VR
    project.
23: pm2 = new ProjectManager("Artifical
    Intelligence","AI");
24: eeTom.addRole(pm2); //Tom leads the AI
    project, too.
```

Figure 3: Sample code for using the basic features of roles.

```
package examples;
import jawiropf.AggregateRole;
public class ProjectManager extends
  AggregateRole {
    String projectName;
    public ProjectManager(String p,String id)
    {  super( id );   projectName = p;   }
   //code for the rest of the class
}
```

Figure 4: Partial code of the `ProjectManager` class.

Lines 21 through 24 demonstrate the aggregate roles. When an aggregate role is coded, the coder must write a constructor with a parameter for the identifier of this instance and call the superclass' constructor. Figure 4 demonstrates this process

Using the Extended Features of Roles

The sample code in Figure 5 demonstrates the use of the extended features in JAWIRO. This code follows the code in Figure 3 as it can be understood from the line numbers.

```
25: eeGordon = new Employee(erMTCX,"628-
    749"); //Gordon is enlisted.
26: eeGordon.suspend(); //Gordon is on
    vacation.
27: eeGordon.resume(); //Gordon is back.
28: pm2.transfer(eeGordon); //Lead of AI
    team is now Gordon.
29: coBML = new Company("Black Mesa Labs",
    "BML");
30: suBML = new Supplier();
31: coBML.addRole(suBML);
32: cuMTCX = new Customer(suBML);
33: coMTCX.addRole(cuMTCX); //MTCX becomes a
    corporate customer of BML.
34: cuTom = new Customer(suBML);
35: peTom.addRole(cuTom); //Tom becomes a
    personal customer of BML.
36: if( coMTCX.canSwitch(
    "examples.Customer") ) {
37:   ((Customer)erMTCX.as(
      "examples.Customer")).buy(3); }
      /* Role switching: Employer->Customer
       * MTCX buys 3 goods */
39: if( peTom.canDelegate(cuTom) ) { //Run-
    time role checking, 2nd way.
40:   cuTom.buy(4); } //Direct execution of
      role,no need for switching as we saw
      peTom plays cuTom.
41: peTom.enableDelegation(true);
42: peTom.useConsultation();
43: ((ProfEmeritus)eeTom.as(
    "examples.ProfEmeritus")).advice();
44: peTom.useDelegation();
45: ((Person)eeTom.as(
    "examples.person")).reportIncident();
46: System.out.println( prTom.bringMember(
    "projectLog") );
47: peTom.executeMethod("advice",null);
48: PersistenceManager pm;
49: pm = new PersistenceManager(
    "C:\\Temp\\","test05");
50: pm.register(peTom,"key_tom");
51: pm.upload(peTom,"key_tom");
```

Figure 5: Using the extended features of roles with Jawiro.

Some of the extended features of roles are quite straightforward to use. Lines 25 through 28 of Figure 5 give examples to such features. The vacation leave of an employee could be coded by letting this person loose this role and then regaining it. However, this coding means that this person is actually fired in the real world. Moreover, resigning from a role also means that its subroles are lost. A suspended role does not loose its subroles and it can easily be suspended. Just like the same fashion, a role can be transferred to another owner with just one command without loosing its subroles. Other extended features of roles need be examined in more detail.

Member Variable and Method Access by Name. This feature makes it possible 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. This feature is implemented in JAWIRO with the `Object bringMember(String name)` method, as seen in line 46 of Figure 5. Although the `String projectLog` member belongs to `ProjectManager` class, it is accessed from a `Person` instance without ever mentioning either the class or the instance which owns the member we need.

If multiple member variables have the same name, the most evolved participant's member is returned. However, dominant participants override this behavior, as examined in the next section. Member variable is returned by value and primitive types are supported only via their respective wrappers. The method `bringLocalMember(String name)` proceeds in the same manner as well, except that it searches for the desired member only within the participant it belongs to.

Member methods are accessed in the same fashion. Line 47 of Figure 5 gives an example for this feature. Unlike member variables, member methods are accessed by reference in JAWIRO and these methods change the internal state of the object they belong to. These rules are dictated by the reflection API of Java, which is used in the `Object executeMethod( String name, Object[] parameters)` and `executeLocalMethod` methods of the `RoleInterface`. The result returned by the executed member method is forwarded via the object returned from these calls.

Dominant Participants in a Role Hierarchy. The methods `bringMember` and `executeMethod` described above search the entire role hierarchy and find the most evolved member. However, this behavior can be overridden by setting some participants as dominant via their `dominateSearch(boolean dominate)` method. When an `Actor` object 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` object acts likewise, but only when the root of its role hierarchy is not also dominant. Otherwise the search order is first the root, then itself, and finally the rest of the role hierarchy. Neither the delegation nor the consultation mechanism, which will be discussed later, affects this procedure.

Multiple Object Level Inheritance. JAWIRO supports multiple object-level inheritance, where owners from different classes are allowed to play the same type of role object. This 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. The `Customer` role of Figure 2 is an example as both `Person` and `Company` instances can have a role of this type. Lines 29 through 35 of Figure 5 set up a scene for demonstrating this feature. A second company, BML, is created and given the supplier role in lines 29-31. Lines 32 and 33 make the previous company, MTCX, a customer of BML. A person, Tom, also becomes a customer of BML in lines 34 and 35. Afterwards, both the person and the company can buy from their common supplier. However, the buying operations are done differently in lines 37 and 40 in order to show the reader the different ways of role checking and role execution.

Implementation of the `Customer` class is given in Figure 6 in order to show how the typing ambiguity is removed by the 'member method access by name' feature.

```
public class Customer extends Role {
  Supplier supplier;
  Object[] params;
  public Customer( Supplier s ) {
    supplier = s; }
  public void changeCustomer(Supplier s) {
    supplier = s; }
  public void buy( int s ) {
    params = new Object[1];
    params[0] = new Int(s);
    supplier.sell( s );
    ((RoleInterface)playedBy())
          .executeLocalMethod
          ("buy",params);
    }}
```

Figure 6: Implementation of a class that uses multiple object level inheritance.

Prevention of Role Binding Anomalies. The following precautions are hard-coded into the role model in order to prevent abnormal role bindings:

- 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 (Gamma, 1994). 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 doesn't approve the operation, the operation is cancelled. The interface that a constraint manager should implement is given in Figure 7.

```
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 );
}
```

Figure 7: The interface for constraint managers.

Delegation or Consultation? By default, JAWIRO works with the consultation mechanism (Bettini, 2003) shown in Figure 8a, 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 8b.

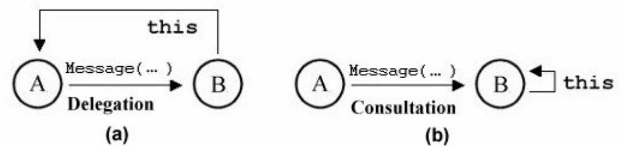

Figure 8: Delegation (a) and consultation (b) mechanisms.

JAWIRO allows switching between consultation and delegation mechanisms at will. 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.

Enabling the delegation mechanism creates a small overhead to the role switching operations of JAWIRO, even if the consultation mechanism is used. Therefore, delegation mechanism is disabled by default. Additionally, execution time of a command is longer with delegation and shorter with consultation. Therefore, users need to explicitly state that they will use delegation instead of consultation in a given time. This is done by first calling the `Actor.enableDelegation(true)` method. Then one can switch to either the delegation mode by the `Actor.useDelegation()` call or to the consultation mode by the `Actor.useConsultation()` call.

Delegation and consultation mechanisms should not be mutually exclusive, as both mechanisms may be needed for better modeling of a real-world system. Lines 41 through 46 of Figure 5 give two examples. The role switching operation in line 43 uses consultation. Both `Employee` and `ProfEmeritus` classes of Figure 2 have a `skill` variable and an `increaseSkill` method. The `advice` method of both classes calls the `increaseSkill`

method, which in turn increases the skill variable. Remember from the code in Figure 3 that our professor, Tom, is also an employee of the company MTCX. Current scenario suggests that Tom 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 Tom's academic skill increased, which is ProfEmeritus.skill. On the other hand, the role operation in line 45 of Figure 5 uses delegation. The Person class of Figure 2 has the reportIncident method, which reports an incident to authorities. This method requires the person to give his/her phone number. If this person is in the office when the incident happens, one needs to switch to Person role from Employee role, to use delegation to report the incident and give the number of the office. Otherwise, home phone would be stated.

Persistency in JAWIRO. 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 Actor each 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, as seen in line 49 of Figure 5. 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, as seen in line 50 of Figure 5. 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 is demonstrated in line 51 of Figure 5. JAWIRO handles the rest of the procedure 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 the encrypted information file, too.
- The PM instance encrypts all created files with the 64-bit DES algorithm.

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 8 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.

```
PersistenceManager pm;
Person peTom = new Person();
pm = new PersistenceManager(
    "C:\\Temp\\","test05");
peTom = (Person) pm.download("key_tom");
```

Figure 8: Sample code for loading a role hierarchy.

Performance of JAWIRO

Although JAWIRO enables better modeling of dynamic systems by implementing all basic and extended features of roles, the performance issues should not be omitted. The extended features of roles are not required by every real world system. Therefore we should investigate whether using the basic features of roles JAWIRO is feasible or not from a performance perspective.

The benchmarking code first creates a role hierarchy with a given depth and degree. The tree representing the hierarchy is a balanced one. 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 performance, we should be able to create hierarchies with arbitrary depth and degree. This need leads to 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. The results of our benchmarks are given in Table 1. They are obtained by using an Intel platform with 2.8GHz Pentium 4 CPU, i865 chipset, 512MB RAM and JDK 1.5.0.

The first stage of the benchmark code is to create the necessary role objects. This is a regular object creation process and not closely related to the performance of JAWIRO.

The second stage is the building phase of the role hierarchy where the role objects created in the first stage are added to appropriate owners so that the resulting tree is balanced. Regardless of the hierarchy depth, JAWIRO introduces virtually zero overhead in this stage.

Overhead of a role checking operation is measured in the third stage. This overhead grows exponentially as the hierarchy depth grows. However, the overhead introduced by JAWIRO is still neglible as it is only 0.055 milliseconds. The results of the fourth stage are similar as well, where the overhead of a role switching command is measured.

The fifth stage of the benchmark measures the overhead introduced by JAWIRO when executing roles. The roles are lightweight for eliminating the execution overhead of the JVM itself. The roles only add a character to the end of a `String` member variable of theirs. The results show that the method execution overhead introduced by JAWIRO is virtually zero and it is independent from the size of the role hierarchy.

Degree=3 (constant)		Average Execution Time (msec.)				# ops.
	Depth	4	5	6	7	
Benchmark Stages	Create members	0.040	0.037	0.045	0.044	n
	Add roles	0.000	0.013	0.013	0.011	n
	Role checking	0.003	0.006	0.018	0.055	n^2
	Role switching	0.004	0.006	0.018	0.055	n^2
	Role execution	0.000	0.003	0.009	0.002	n
	Switching execution	0.002	0.007	0.023	0.075	n^2
	Checking switching execution	0.003	0.008	0.038	0.099	n^2
	n =	39	120	363	1092	
	n^2 =	1521	14400	131769	$1.2x10^6$	

Table 1: Benchmark results. n represents the number of the role objects in the hierarchy and #ops represents how many times the command is executed.

Overhead of role switching and execution is measured in the sixth stage. This means the execution time of a role with an `as` command, similar to line 11 in Figure 3. The results of this stage are exponential like the third stage and it is still a neglible value.

The latest stage of the benchmark measures the overhead of role checking, switching and execution, similar to lines 10 and 11 in Figure 3. Even with more than one million operations in a role hierarchy that contains 1092 role objects, the overhead introduced by JAWIRO is less than one percent of a millisecond.

Related Work

As role models proved themselves valuable in modeling dynamic systems, there is a considerable amount of work in the literature. This section gives a brief overview of recent role models.

INADA (Aritsugi, 2000) is an extension of C++ with role support in a persistent environment where every type is equal to a role. The limitations of INADA are its inability to support aggregate roles and the non-existence of methods for run-time type control. Moreover, Roles are presented in a set based fashion, which is a weaker representation than the relational hierarchy of roles. Other basic features of roles are supported by INADA.

DEC-JAVA (Bettini, 2003) bears inspirations from the decorator design pattern (Gamma, 1994) and adds its own syntax for declaring roles. DEC-JAVA relies on dynamic specialization of methods and it is still in prototype stage.

The role model proposed by Schrefl and Thalhammer (Schrefl, 2004) is an elegant and flexible one that supports all primary characteristics of roles. Using the basic features of roles in JAWIRO is similar with Schrefl's model. However, the extended features discussed in previous section are not available in Schrefl's role package. This role model is based on a previous work in Smalltalk (Gottlob, 1996).

The focus of Lee and Bae's work (Lee, 2002) is preventing the violation of structural constraints and abnormal role bindings. When a core (actor) object has multiple roles, individual role objects are grouped into one big, composite role. This prevents a hierarchical relation between roles. Moreover, supporting aggregate roles in Lee and Bae's model is impossible. The final drawback of that model is the missing *Select* composition rule. When there are name conflicts (a primary feature 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.

	JAWIRO	(Schrefl, 2004)	(Aritsugi, 2000)	(Bettini, 2003)	(Lee, 2002)
Base language	Java	Java	C++	Java	Java
Aggregate roles	+	+	–	+	–
Hierarchy support	+	+	–	+	–
Run-time role checking	+	+	–	–	–
Object level multiple inheritance	+	–	–	–	–
Member/method access without referring its owner.	+	–	–	–	–
Preventing role binding anomalies	+	–	–	–	+
Persistency	+	–	+	–	–

Table 2: Feature based comparison of recent role models.

Results and Future Work

JAWIRO is currently, to the best of our knowledge, the only role model which has complete support for both the basic and the extended features of roles. Table 2 gives a feature comparison of the current role models introduced in the previous section. On the performance side, the execution overhead introduced by JAWIRO is virtually zero. JAWIRO role model is available for download from http://www.yunusemreselcuk.com/jawiro/index.html. The API documentation and some examples are also available at this URL.

Future work will be determined mostly by the needs which will arise when JAWIRO is incorporated in current and future software projects. One particular feature of JAWIRO which is open for future enhancements is the persistency feature. Currently JAWIRO uses unorthogonal persistency; using orthogonal persistency will introduce benefits such as simpler semantics and incremental evolution (Atkinson, 1995).

References

Aritsugi, M., and Makinouchi, A. 2000. Multiple-Type Objects in an Enhanced C++ Persistent Programming Language. *Software - Practice and Experience*. 30/2: 151–174

Atkinson, M., and Morrison, R. 1995. Orthogonally Persistent Object Systems. *VLDB Journal*. 4: 319–401

Bettini, L., Capecchi, S., and Venneri, B. 2003. Extending Java to Dynamic Object Behaviours. *Electronic Notes in Theoretical Computer Science*. 82/8

Drossopoulou, S., Damiani, F., and Dezani-C., M. 2002. More dynamic object reclassification: Fickle. *ACM Trans. Programming Languages and Systems*, 2:153-191.

Fowler, M. Dealing with Roles. *Unpublished paper*. In http://martinfowler.com/apsupp/roles.pdf

Gamma, E. et al. 1994. *Design patterns elements of reusable object oriented software.* AddisonWesley.

Gottlob, G., Schrefl, M., and Röck, B. 1996. Extending object-oriented systems with roles. *ACM Trans. on Information Systems*. 14/3:268–296

Kristensen, B.B. 1996. Conceptual Abstraction Theory and Practical Language Issues. *Theory and Practice of Object Systems*. (2)3

Lee, J-S., and Bae, D-H. 2002. An enhanced role model for alleviating the role-binding anomaly. *Software - Practice and Experience,* 32:1317–1344.

Schrefl, M., and Thalhammer, T. 2004. Using roles in Java. *Software-Practice & Experience*. (34): 449–464.

Selcuk, Y.E., and Erdogan, N. 2003. How to solve the inefficiencies of object oriented programming: A survey biased on role-based programming. In *7th World Multiconf. on Systemics, Cybernetics and Informatics*.

Selcuk, Y.E., and Erdogan, N. 2004. JAWIRO: An Extended Role Model for Java. In *Int'l. Conf. on Computational Intelligences*.

Ungar, D., and Smith, R.B. 1987. Self: The power of simplicity. In *Proc. ACM Conf. on Object Oriented Programming Systems, Languages and Applications*.

Wong, R.K., et. al. 1997. A Data Model and Semantics of Objects with Dynamic Roles. In *IEEE Int'l Conf. On Data Engineering*.

Zendler, A.M. 1998. Foundation of the Taxonomic Object System. *Information and Software Technology*. 40:475-492.