Introducing Ontologically Founded Roles in Object Oriented Programming: powerJava

Matteo Baldoni and **Guido Boella**
Dipartimento di Informatica. Università di Torino - Italy
baldoni@di.unito.it, guido@di.unito.it

Leendert van der Torre
CWI Amsterdam and TUDelft
torre@cwi.nl

Abstract

In this paper we introduce a new view on roles in Object Oriented programming languages. Our notion is based on an ontological analysis of social roles and attributes to roles the following properties: first, a role is always associated not only to an object instance playing the role, but also to another object instance which constitutes the context of the role and which we call institution. Second, the definition of a role depends on the definition of the institution which constitutes its context. Third, this second property allows to endow players of roles with powers to modify the state of the institution and of the other roles of the same institution. As an example of this model of roles in Object Oriented programming languages, we introduce a role construct in Java.

Introduction

The concept of role is used quite ubiquitously in Computer Science: from databases to multiagent systems, from conceptual modelling to programming languages. According to (Steimann 2000), the reason is that even if the duality of objects and relationships is deeply embedded in human thinking, yet there is evidence that the two are naturally complemented by a third, equally fundamental notion: that of roles. Although definitions of the role concept abound in the literature, Steimann maintains that only few are truly original, and that even fewer acknowledge the intrinsic role of roles as intermediaries between relationships and the objects that engage in them. There are three main views:

- Names for association ends, e.g., in UML and Entity-Relationship diagrams.
- Dynamic specialization, like in the Fibonacci (Albano *et al.* 1993) programming language.
- Adjunct instances, like in the DOOR programming language (Wong, Chau, & Lochovsky 1997).

The two last views are more relevant for modelling roles in programming languages. Roles are modelled in (Albano *et al.* 1993) by dynamically reclassifying an object, while in (Wong, Chau, & Lochovsky 1997) they are seen as special instances of a separate kind of classes which are associated with traditional object instances.

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

Both of them have pros and cons. For example, dynamic specialization captures the dynamic relation between a class and a role which can be played by it (e.g., a person can become a student), but it less easily models the intuition that roles can have their own state (e.g., an employee has a different phone number than the person playing that role). In contrast, roles as adjunct instances can obviously have their own state, but they may pose problems when role instances are detached from the object which plays the role.

There is a wide literature on the introduction of the notion of role in programming languages. However, most works extend programming languages with roles starting from practical considerations, starting from (Bachman & Daya 1977) who use roles to reduce the heterogeneity of a set of records. In contrast, the research question of this paper is the following: How to introduce in an Object Oriented programming language a notion of role which is ontologically well founded? Our ontological analysis of the notion of role is made in (Boella & van der Torre 2004c; 2004a; 2004b). In our model features which are sometimes already present in previous models receive a new interpretation under the light of a well founded underlying conceptual model.

The methodology we follow is to introduce a new programming construct in a real programming language, Java, since it is one of the most used Object Oriented languages and one of the most principled. To prove its feasibility, we translate the new language, called powerJava, to pure Java by means of a precompilation phase. It is beyond the scope of this paper to provide a formal semantics of the new construct or to define the associated type theory. Moreover we do not address other issues related to roles like the problem of method delegation (Albano *et al.* 1993) or of roles playing other roles (Wong, Chau, & Lochovsky 1997), but we leave them for future work.

The role construct we introduce in Java promotes the separation of concerns between the core behavior of an object and its context dependent behavior. In particular, the interaction among a player object, the institution and the other roles is encapsulated inside the role the object plays.

First we present our ontological definition of roles and then we introduce roles in Java with powerJava and we describe how powerJava can be translated in pure Java. Conclusion ends the paper.

Properties of roles

Consider as a running example the roles student and teacher. A student and a teacher are always a student and a teacher of some school. Without the school the roles do not exist anymore: e.g., if the school goes bankrupt, the actors (e.g., persons) of the roles cannot be called teachers and students anymore. The institution (the school) also specifies the properties of the student, which extend the properties of the person playing the role of student: the school specifies its enrolment number, its scores at examinations, and, above all, also how it can behave as a student. For example, the student can take an exam by submitting some written examination. A student can make the teacher evaluate its examination and register the mark because the school defines both the student role and the teacher's role: the school defines how an examination is evaluated by a teacher, and maintains the official records of the examinations. Otherwise the student could not have an effect on the teacher. But in defining such actions the school *empowers* the person who is playing the role of student: This example highlights the following distinguishing properties of roles (Boella & van der Torre 2004c; 2004a; 2004b):

- *Foundation*: a (instance of) role, besides being associated with an instance of its player (see (Guarino & Welty 2002)), must always be associated with an instance of the institution it belongs to,
- *Definitional dependence*: the definition of the role must be given inside the definition of the institution it belongs to. This is a stronger version of the definitional dependence notion proposed by (Masolo *et al.* 2004), where the definition of a role must use the concept of the institution.
- *Institutional empowerment*: the actions associated with the role in the definition of the institution have access to the state and actions of the institution and of the other roles: they are powers.

Moreover, as (Guarino & Welty 2002) notice, contrary to natural classes like person, roles lack of *rigidity*: a player can enter and leave a role without losing its identity; a person can stop being a student but not being a person.

Finally, in this paper, we consider also (Steimann 2000)'s analysis of roles. It highlights the polymorphism inherent to roles: a role can be played by different kinds of actors. For example, the role of customer can be played by instances both of person and of organization, i.e., two classes which do not have a common superclass. The role must specify how to deal with the different properties of the possible actors. This requirement is in line with UML modelling language, which relates roles and interfaces as partial descriptions of behavior. This last property compels to avoid modelling roles as dynamic specializations as, e.g., (Albano *et al.* 1993; Gottlob, Schrefl, & Rock 1996) do. If customer were a subclass of person, it could not be at the same time a subclass of organization, since person and organization are disjoint classes. Symmetrically, person and organization cannot be subclass of customer, since a person can be a person without ever becoming a customer.

Introducing roles in Java

Roles are useful in programming languages for several reasons, from dealing with the separation of concerns between the core behavior of an object and its interaction possibilities, to reflecting the ontological structure of domains where roles are present (Baldoni, Boella, & van der Torre 2005a), from modelling dynamic changes of behavior in a class to fostering coordination among components (Baldoni, Boella, & van der Torre 2005b).

In this section we discuss how our ontological definition of role (Boella & van der Torre 2004c; 2004a; 2004b) can be introduced in Object Oriented programming languages. As noticed, modelling roles as dynamic specializations does not capture the idea that roles have their own state and limits their polymorphism. Hence, we model roles as instances of role classes, which can be associated at runtime to objects which can play a role. However, roles are a special kind of objects, and instances of role classes do not exist on their own, but they always require to be associated with an object instance of its player and an object instance of the related institution. The relations of a role with these two instances are different. Concerning the former relation, the player of the role is an object whose properties and behavior are extended when it is seen under the perspective of the role. Moreover, the role does not affect the core behavior of the player, but the behavior of the role is determined in part by its player. To do so, a player of a role is required to have a certain behavior, i.e., methods. In contrast, concerning the latter relation, the object instance which represents the institution which the role belongs to gives the role powers: the role is enabled to access the institution's own state and the state of other roles via its methods; thus, the role's behavior can affect the institution's behavior. Accessing the institution state is possible only if the class defining the institution and the class defining the role are connected. This is what we call definitional dependence and it requires that the role class belongs the namespace of the institution class.

In the same way as classes are distinguished from interfaces, we distinguish the class implementing a role in an institution from the role definition which specifies its abstract behavior and which objects can play the role. In this way we represent that giving exams is a feature of students, independently from the specific class of institution the student is a student of. In order to capture such generalization of a role with respect to the institution it belongs to, the definition of a role should be kept distinct from the implementation of a role in a given type of institution. Moreover, such institution when it implements the role definition can further specify the role by adding fields and further methods. Playing a role does not require any special preparation of the class whose instances want to play a role, apart from exhibiting some requested behavior, i.e., the class has to implement some methods requested by the role as specified, e.g., by an interface. In this way the implementation of the role and the definition of the players can be developed independently.

Finally, the constraint of foundation requires that the creation of a role instance involves both an institution instance and an object instance, which must be specified in some way as arguments. When a role is instantiated and its player

is associated with it, the methods representing the powers of the roles can be invoked on its player object. To avoid method clashes and preserve a sound type system, differently from other approaches, a power can be invoked from a role player only by specifying the role which the player has to play. Note that an object can play not only several roles, but also the same role in different institutions at the same time. Hence, the role under which a player is seen is specified using not only the role but also the institution instance.

In summary, first a role is defined specifying what is requested to play a role and what is offered by a role. Second, the role definition is implemented in a class connected to the institution class the role belongs to. Finally, the role implementation class is instantiated connecting the player instance with the institution instance. At this point, powers can be invoked on the player. However, the extension of an object with roles is transparent to the programmer, who sees roles only as perspectives on the player objects and he needs not a reference to the role instance itself to invoke its powers.

In this paper we map in Java these desired features of roles in Object Oriented programming languages, taking advantage from the existing features of the language when possible. First, when there is the need to specify abstract behavior we use the notion of interface in Java, like in the case of specifying the powers of a role and the methods required to play a role. Second, since inner classes in Java allow a class to belong to the namespace of another class, we use them to give powers to roles in institutions. Moreover, implementing a role definition as an inner class of an outer class defining an institution parallels exactly the definitional dependence. Third, the association of a role instance with an institution instance can be dealt with the implicit reference in Java of an inner class from its outer class. So we are left only to deal explicitly with the association of a role instance with a player instance, to complete foundation. Finally, seeing an object under a role is paralleled with type casting in Java.

In the next sections we will detail this mapping. Since powers are a distinguishing feature of roles in our model, we call our language powerJava.

The definition of roles

The definition of a role has to specify both what is required to play the role and which powers its players have in the institutions in which the role will be implemented. In order to make role systems reusable, it is necessary that a role is not played by a class only. For (Steimann & Mayer 2005), roles define a certain behavior or protocol demanded in a context independently of how or by whom this behavior is to be delivered. Thus, roles must be specified independently of the particular classes playing the role, so that the objects which can play the role might be of different classes and can be developed independently of the implementation of the role. This is a form of polymorphism. To achieve such polymorphism we associate with a role partial descriptions of classes listing the signatures of the methods which are requested to an object in order to play a role. Thus, roles make no assumption about how a certain piece of functionality is achieved: they allow for unrelated types taking the same role.

We thus have that a role definition must express two faces:

- The methods required from objects playing the role: *requirements*. For the instances of a class to play a role, the class must offer these methods. These are specified by the role as an interface.
- The methods offered to objects playing the role: *powers*. If an object of a class, offering the required methods, plays the role, it is empowered with these new methods.

This double face pervades the life of a role: first, a role is defined with its requirements and powers, then its powers are implemented in a class which connects a role with a player satisfying its requirements; finally, the class implementing the role is instantiated passing as argument to its constructor an instance of an object satisfying the requirements.

The definition of a role using the keyword `role` in Figure 1 is similar to the definition of an interface; it is the specification of the powers acquired by the role in the form of abstract methods signatures (`interfacebody`). The only difference is that the role definition by means of the keyword `playedby` refers also to another interface, that in turn specifies the requirements which an object playing the role must satisfy. This mechanism mirrors the idea that *roles have two faces*: the *requirements* and the *powers*. Moreover, differently than an interface, also static variables cannot be declared. Thus, nothing is required for an object to become the player of a role, apart implementing the appropriate behavior required by the role as specified by the requirement interface. In the example of Figure 3, `role` specifies the powers of `Student`, whilst the interface `StudentReq` specifies its requirements. The implementation of the requirements is given inside the class of the object playing the role. The implementation of the powers is given in the definition the class defining the institution the roles belong to.

Institutions and definitional dependence

In our model roles are always associated to an instance of, and are definitionally dependent on, an institution. The term "definition" of an object refers to the definition of the class the object is an instance of; thus, the class defining an institution includes the definition of the class implementing the roles of the institution. How do these properties reflect on which kind of methods can be invoked on a role? For the notion of role to be meaningful, these methods should go beyond standard methods whose implementation can access

```
roledef ::= "role" identifier
   ["extends" identifier*]
   "playedby" identifier interfacebody

roleimpl ::= [public | private ] [static]
   "class" identifier ["realizes" identifier]
   ["extends" identifier]
   ["implements" identifier*] classbody

rcast ::= (expr "." identifier) expr
```

Figure 1: The extension of the Java syntax in powerJava.

the private state of the role only. Rather, roles add powers to objects playing the roles. Power means the possibility to modify also the state of the institution which defines the role and the state of the other roles defined in the same institution. In our running example, we have that the method for taking an exam in the school must be able to modify the private state of the school. E.g., if the exam is successful, the grade should be added to the registry of exams in the school. Analogously, the student's method for taking an exam can invoke the teacher's method of evaluating an examination.

Powers, thus, seem to violate the standard encapsulation principle, where the private variables are visible to the class they belong to only. However, here, the encapsulation principle is preserved: all roles of an institution depend on the definition of the institution; so it is the institution itself which gives to the roles access to its private fields and methods, thus allowing coordination inside the institution. Since it is the institution itself which defines its roles, there is no risk that a role abuses of its access possibilities.

Enabling a class to belong to the namespace of another class without requiring it to be defined as friend (like C++ does at the cost of endangering encapsulation) is achieved in Java by means of the inner class construct. Thus, we extend the notion of inner class to allow roles to be implemented inside an institution (`roleimpl`). The inner class construct is extended with the keyword `realizes` which specifies the name of the role the inner class is implementing. An institution is simply a class with an inner class implementing roles. In the example of Figure 4, `StudentImpl` implements the role definition `Student`, inside the institution `School`.

In powerJava, an inner class that realizes a role must implement the corresponding role definition in the very same way as a class implements an interface and it can add (non-static) fields and further (non-static) methods. Moreover, since a role implementation is a (inner) class, it can be an institution itself with its own role implementations, it can enact other roles, it can extend other classes (unless they realize roles), implement interfaces, *etc.* Analogously, the institution is a class which can play a role, *etc.*

Since the behavior of a role instance depends on the player of the role, in the method implementation, the role player instance can be retrieved via a new reserved keyword: `that`. So this keyword refers to *that* object which is playing the role at issue, and it is visible only in the role implementation. The value of `that` is initialized when the constructor of the role implementation is invoked. The referred object has the type defined by the role requirements or a subtype.

We do not need a special expression for creating instances of the inner classes implementing roles, because we use the Java inner classes syntax: starting from an institution instance, the keyword `new` allows the creation of an instance of the role as an instance of the inner class (e.g., `harvard.new StudentImpl(chris)`). Note, however, that it is not possible to instantiate a role definition (in the same way as it is not possible to instantiate an interface): what is instantiated is always the implementation of a role in an institution. Moreover, roles can be implemented in different ways in the same institution. For example, the class `School` could implement the role definition `Student` as both `FirstYearStudentImpl` and `SecondYearStudentImpl`.

All the constructors of all role implementations have an implicit first parameter which must be bound to the player of the role and becomes the value of `that`. The reason is that to construct a role instance we need both the institution the role belongs to (the object the construct `new` is invoked on) and the player of the role (the implicit first parameter). For this reason, the parameter has as its type the requirements of the role: e.g., the constructor `StudentImpl` has an implicit first parameter of type `StudentReq`. This first parameter must not be declared explicitly in the implementation of the role, but it is added by the precompiler. For example, let us suppose that `harvard` is an instance of `School` and that `chris` is a person who wants to became a student of `harvard`. The instruction `harvard.new StudentImpl(chris)` expresses this fact, given that `StudentImpl` is an inner class that implements the role `Student` inside the institution `School`. This expression returns an object of type `School.StudentImpl`. In powerJava, the constructor of a role implementation must always have a first parameter which is used to pass to the role instance the value of the player of the role.

Exercising the powers of a role

A role represents a perspective on an object. An object has different (or additional) properties when it is seen in the perspective of a certain role, and it can perform new activities, which we call powers, as specified by the role definition. In (Steimann 2000)'s terminology, a role is a sort of type specifying behavior.

When an object is seen under the perspective of a role, we want that the object has a specific state for it. This state is different from the player's one, it is specific to each role in each institution, and it can evolve with time by invoking methods on the role (or on other roles of the same institution as we have seen in the running example). This state is maintained by a role instance which is associated to the player. Since a role represents the perspective on an object, the object playing it should be able to invoke the role's methods without making any explicit reference to the instance of the role: in this way the association between the object instance and the role instance is transparent to the programmer. The object specifies only in which role it is invoking the method. For example, if a person is a student and a student can be asked to return its enrollment number, then we want to be able to invoke the method on the person as a student without referring to the student role instance.

The same methods will have a different behavior according to the role which the object plays when they are invoked. On the other hand, methods of a role can exhibit different behaviors according to who is playing it. So a method of student returning the name of the student together with the name of the school returns different values for the name according to whom is playing the role of student. This is possible since the implementation of methods representing powers uses the methods required by the role to its player in order to play the role via the `that` keyword. These required methods obviously can access the state of the player since

they are part of the implementation of the player.

In our model, roles are always roles in an institution. Hence, an object can play at the same moment the same role more than once, albeit in different institutions. For example, one can at the same time be a student at the high school, a student of foreign languages in another school, *etc.* So a method returning the name of the student together with the name of the school returns a different name of school according to which school the student role is played in. An object can play several roles in the same institution. E.g., a person can be an MP and a minister at the same time (even if it is not required to be an MP to become minister).

In order to specify the role under which an object is referred, we evocatively use the same terminology used for casting by Java. For example, if a person is playing the role of student and we want to invoke a method on it as a student, we say that there is a casting from the object to the role. To refer to an object in a certain role, both the object and the institution where it plays the role must be specified. We call this methodology *role casting*. Type casting in Java allows to see the same object under different perspectives while maintaining the same structure and state. In contrast, role casting views an object as having a different state and different behaviors when playing different roles. This is because it conceals a *delegation* mechanism: the player instance has a hidden delegation to the role instance the execution of the method. The delegated object can only act as allowed by the powers of the role, but it can access the state of the institution and, by exploiting the construct `that`, it can also refer to the delegating object to invoke the requirements. Role casting allows to make transparent to the programmer the association of a role and an object instance: the programmer invokes a method of a role on the object playing it casted into the role; the language transforms this method invocation in a message sent to the delegated role instance, which is hidden in its player.

So, the last syntactic change in powerJava is the introduction of *role casting expressions* extending the original Java syntax for casting. A `rcast` specifies both the role and the instance of the institution the role belongs to, e.g.: in `(harvard.TeacherImpl) george` the person `george` is casted to its role `harvard.TeacherImpl` of type `School.TeacherImpl`.

Figure 2 reports the relations among the main concepts, in the form of a simple UML diagram. Requirements of a role (*Role Requirement*) correspond to an interface, specifying which methods must be defined in a class whose instances play the role. Powers are a new concept proper of the `role` construct: they are the specification of the new methods (*Role Power*). *Role Power Implementation* is specified as an inner class of *Institution*. Inner classes express the fact that the namespace of the institution is visible from the role implementation: i.e., the institution defines a namespace which is shared by all the roles The fact that inner classes belong to the namespace of the outer class in UML is represented by the arrow with a plus sign within a circle at the end attached to the namespace. An inner class that implements the power of a role has always two references to two objects: the institution that defines it and the player that plays it.

Using powerJava

In Figures 3–6, we present our running example in powerJava. In Figure 3, the definitions of the roles `Student` and `Teacher` are introduced. The role definitions specify, like an interface, the signatures of the methods that correspond to the powers that are assigned to the objects playing the role. For example, returning the name of the `Student` (`getName`), submitting a homework as an examination (`takeExam`), and so forth. Moreover, we couple a role definition with the specification of its requirements by the keyword `playedby`. This specification is given by means of the name of a Java interface, e.g., `StudentReq`, imposing the presence of methods `getName` and `getSocialSecNumber`).

As explained, role definitions must be implemented inside institutions. In our example, the role `Student` is implemented as an inner class of `School` called `StudentImpl` which `realizes Student` (see Figure 4). Being an instance of a class, a role instance has a state, specified by its private fields, in this example, `studentID`. It is worth noticing that the method `getStudentID` is not part of the role definition `Student` in Figure 3. Powers have access to private variables and methods of the institution and of the sibling roles: e.g., the method `takeExam` directly accesses the private variable `mark` of the institution, and the method `evalHomeWork` of the teacher.

The method `evalHomeWork` of `TeacherImpl` in Figure 4 deserves a remark. Such a method is invoked by the method `takeExam` in `StudentImpl`, that records a mark in the registry of the school. Of course, the role teacher has not the actual ability to perform the evaluation of the homework of a student, this ability is a feature of the `Person` that plays the role of the teacher. For this reason `evalHomeWork` uses the method `read` of the object referenced by `that`, which plays the role of the teacher.

This pattern, in which the methods of the actual player of a role that are specified by the requirements are invoked resembles the OBSERVER pattern, where the player is the observer. The observer contains a set of behaviors that are invoked as a consequence of an event, that occurred in the observable (in our case the corresponding role). The difference is that the observable role is plugged inside an institution that realizes the coordination of the events that can occur (and, so, coordinates the reactions of the players).

Moreover, this pattern allows a form of exogenous coordination (Baldoni, Boella, & van der Torre 2005b): the `Persons` in the `Teacher` and `Student` role are simply components whose behavior is used at the right moment without requesting it of being aware of which is the student taking the examination. Symmetrically, the person playing the role is a component which does not know anything about which teacher will evaluate his homework. Connecting the two components is responsibility of the institution which works as a coordinator as in the IWIM model of (Arbab 1996; Guillen-Scholten *et al.* 2003).

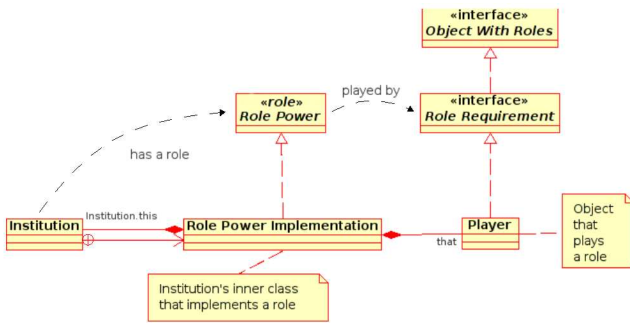

Figure 2: Roles and Institutions in UML.

In order for an object to play a role it is sufficient that it conforms to the role requirements. Since the role requirements are a Java interface, it is sufficient that the class of the object implements the methods of such an interface. In Figure 5, the class `Person` can play the role `Student`, because it conforms to the interface `StudentReq` by implementing the methods `getName` and `getSocialSecNumber`.

A role instance is created by means of the construct `new` and by specifying the name of the inner class implementing the role. In Figure 6, the object referred by `chris` can be assigned the role of a student of the school `harvard` by executing the following instruction:
`harvard.new StudentImpl(chris)`

In this context, i.e., within the role implementation, the keyword `that` will refer to `chris`.

Moreover, note that the same person can play the same role in more than one school. In the example `george` is both a teacher of `mit`, `mit.new TeacherImpl(george)`, and a teacher of `harvard`, `harvard.new TeacherImpl(george)`.

It is not necessary to assign to a variable the object returned by the constructor. Differently from other objects, role instances do not exist by themselves and are always associated to their players: when it is necessary to invoke a method of the student it is sufficient to have a referent to its player object. However, it is possible to assign role instances to variables as in:
`Student st2 = sc2.new MyStudent(pe2);`

In this case the variable `st2` is typed with the rolename, as it is possible for interfaces. The role instance is thus upcasted to the role type and the methods specific to the class `MyStudent` are not visible anymore. In this way the variable `st2` can be assigned students of whatever type of institution implements them (e.g., we can have a driving school class which implements the role `Student`).

The player of a role is involved in the instantiation of the role and, as soon as such instance is created, the player can start exercising its powers. Differently than other objects, role instances do not exist by themselves and are always associated to their players: when it is necessary to invoke a method of the student it is sufficient to have a reference to its player object. For this reason, it is not necessary to put the result of the `new` operation in a variable. Methods can be invoked directly from their players, given that the player is seen in its role (e.g., `StudentImpl`). This is done in powerJava by casting the player of the role to the role implementation we want to refer to.

Since roles do not exist out of an instance of the institution defining them, in order to specify a role, it is necessary to specify the institution it belongs to. In the syntax of powerJava the structure of a role casting is captured by the construct `rcast` shown in Figure 1. For instance, `((harvard.TeacherImpl) george).getName()` takes `george` in the role of teacher in the institution `harvard`. As a result, if `getName` applied to `george` returns only the person's name (`george.getName()`), with the cast, the same invocation will return "George, teacher at Harvard". Obviously, if we cast `george` to the role of teacher at `mit` `((mit.TeacherImpl) george).getName()`, we obtain "George, teacher at MIT".

With respect to type casting, role casting does not only select the methods available for the object, but it views the object as having a different state and its methods as having a different meaning: the name returned by the role is different from the name of the player since the method has a different behavior. It is important to observe that role casting is done to the inner class implementing the role but the role instance can always be type casted to the role definition as well as it can be done with Java interfaces: `((Teacher)(harvard.TeacherImpl) george).getName()`.
While in the previous case it was possible to use all the methods of the specific implementation, in this case, only the methods that are specified in the role definition can be applied. E.g., the method `getTeacherID` and `getStudentID` are not accessible because they are not part of the role definitions `Teacher` and `Student`, respectively. Hence, as it is done in Java for the interfaces, role definitions can be viewed as types, and, as such, they can be used also in variable declarations, parameter declarations, and as method return types. Thus, roles allow programmers to conform to (Gamma *et al.* 1995)'s principle of "programming to an interface". It is possible to define variables with a role as their type, with assigned as values objects with as types role implementations of that role belonging to the same or to different institution classes.

Finally, powerJava allows the implementation of roles which can be further articulated into other roles. For example, a school can be articulated in teaching classes (another social entity) which are, in turn, articulated into student roles. In this way, it is possible to create a hierarchy of social entities, where each entity defines the social entities it contains, like proposed by (Boella & van der Torre 2004b).

```
interface StudentReq
{ String getName();
  int getSocialSecNumber(); }

role Student playedby StudentReq
{ String getName();
  void takeExam(int examCode, HomeWork hwk);
  int getMark(int examCode); }

interface TeacherReq
{ String getName();
  int getSocialSecNumber(); }

role Teacher playedby TeacherReq
{ String getName();
  int evalHomeWork(HomeWork hwk); }
```

Figure 3: Definition of roles and their requirements.

Translating roles in Java

In this section we summarize the translation of the role construct into Java, for giving a semantics to powerJava and to validate our proposal. This is done by means of a precompilation phase. The precompiler (available at `http://www.powerjava.org` together with the complete translation of the above examples) has been implemented by means of the tool javaCC, provided by Sun Microsystems.

The role definition is simply an interface to be implemented by the inner class defining the role. So the role powers and its requirements form a pair of interfaces used to match the player of the role and the institution the role belongs to. The relation between the role interface and the requirement interface is used in the constructor of an inner class implementing a role.

When an inner class implements a role, the role specified by the `realizes` keyword is simply added to the interfaces implemented by the inner class. The correspondence between the player and the role object, represented by the construct `that`, is precompiled in a field called `that` of the inner class. This field is automatically initialized by means of the constructors which are extended by the precompiler by adding a first parameter to pass the suitable value. The constructor adds to its player referred by `that` also a reference to the role instance (`setRole`, see below). The remaining link between the instance of the inner class and the outer class defining it is provided automatically by the language Java (`School.this` in our running example).

To play a role an object must be enriched by some methods and fields to maintain the correspondence with the different role instances it plays in the different institutions. This is obtained by adding, at precompilation time, to every class a structure for bookkeeping its role instances. This structure can be accessed by the methods whose signature is specified by the `ObjectWithRole` interface. Since every object can play a role, it is worth noticing that the ideal solution would be that the `Object` class itself implements `ObjectWithRole`. The two methods that are introduced by the precompiler are `setRole` and `getRole` which respectively adds a role to an object, specifying where the role is played, and returns the role played in the institution passed as parameter. Further methods can be added for leaving a role, transferring it, *etc.*

```
class School {
  private int[][] marks;
  private Teacher[] teachers;
  private String schoolName;

  class StudentImpl realizes Student {
    private int studentID;
    public int getStudentID()
        { return studentID; }
    public void takeExam(int examCode;
                         HomeWork hwk)
      { marks[studentID][examCode] =
        teachers[examCode].evalHomeWork(hwk); }

    public String getName()
    { return that.getName() +
      ", student at " + schoolName; }
  }

  class TeacherImpl realizes Teacher {
    private int teacherID;
    public int getTeacherID()
        { return teacherID; }
    public int evalHomeWork(HomeWork hwk)
        { ... return mark; ... }
    public String getName()
        { return that.getName() +
          ", teacher at " + schoolName; }
  }
}
```

Figure 4: An institution and its role implementations.

We present one possible implementation of these methods which is supported by a private hashtable `rolelist`. As key in the hashtable we use the institution instance address and the name of the inner class. As an example, the class `Person` plays the role `Teacher`. So its instances will have a hash-table that keeps the many roles played by them. In the case of `george` there will be two role instances: he is a teacher of `harvard` and of `mit`.

Role casting is precompiled using these methods. The expression referring to an object in its role (a `Person` as a `Teacher`, e.g., `(harvard.TeacherImpl) george`) is translated into the selector returning the reference to the inner class instance, representing the desired role with respect to the specified institution. The translation will be `george.getRole(harvard, "TeacherImpl")`. The string `"TeacherImpl"` is provided because in our solution it is used as a part of the key of the hashtable.

```
class Person implements StudentReq,TeacherReq
{
  private String name;
  private int socialSecNumber;
  String getName()
     { return name; }
  int getSocialSecNumber()
     { return socialSecNumber; }
}
```

Figure 5: Players of roles.

```
Person  chris = new Person("Christine");
Person  george = new Person("George");
School  harvard= new School("Harvard");
School  mit = new School("MIT");

harvard.new StudentImpl(chris);
harvard.new TeacherImpl(george);
mit.new TeacherImpl(george);

((harvard.StudentImpl) chris).getName();
((harvard.TeacherImpl) george).getName();
((Teacher)(mit.TeacherImpl) george).getName();
((harvard.StudentImpl) chris).takeExam(...,...);
```

Figure 6: Using roles.

Conclusions

In this paper we introduce a new view on roles in Object Oriented programming languages based on an ontological analysis of the notion of role. We introduce this model of roles in the new programming language powerJava, which extends Java. First, roles are always associated not only to an object instance playing the role, but also to another object instance which constitutes the context of the role and which we call institution. Second, the definition of a role depends on the definition of the institution which constitutes its context. Third, this second property allows to endow players of roles with powers to modify the state of the institution and of the other roles of the same institution.

Future work concerns directly compiling powerJava in the Java bytecode, building a formal semantics of the new constructs and defining the associated type system. Moreover, the role construct can be enriched with the possibility to extend roles, to suspend and transfer roles, and other features. Finally, powerJava is used as a coordination language dealing with concurrency, given its ability to deal with the separation of concerns (Baldoni, Boella, & van der Torre 2005b) and to model multiagent systems (Baldoni, Boella, & van der Torre 2005a).

References

Albano, A.; Bergamini, R.; Ghelli, G.; and Orsini, R. 1993. An object data model with roles. In *Procs. of VLDB'93*, 39–51.

Arbab, F. 1996. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models, COORDINATION '96*, volume 1061 of *LNCS*, 34–56. Springer.

Bachman, C., and Daya, M. 1977. The role concept in data models. In *Procs. of VLDB'77*, 464–476.

Baldoni, M.; Boella, G.; and van der Torre, L. 2005a. Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *Procs. of PROMAS'05 workshop at AAMAS'05*.

Baldoni, M.; Boella, G.; and van der Torre, L. 2005b. Roles as a coordination construct: Introducing powerJava. In *Procs. of MTCoord'05 workshop at COORDINATION'05*.

Boella, G., and van der Torre, L. 2004a. An agent oriented ontology of social reality. In *Procs. of FOIS'04*, 199–209. Amsterdam: IOS Press.

Boella, G., and van der Torre, L. 2004b. Organizations as socially constructed agents in the agent oriented paradigm. In *LNAI n. 3451: Procs. of ESAW'04*, 1–13. Berlin: Springer Verlag.

Boella, G., and van der Torre, L. 2004c. Regulative and constitutive norms in normative multiagent systems. In *Procs. of KR'04*, 255–265. AAAI Press.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Software*. Addison-Wesley.

Gottlob, G.; Schrefl, M.; and Rock, B. 1996. Extending object-oriented systems with roles. *ACM Transactions on Information Systems* 14(3):268 – 296.

Guarino, N., and Welty, C. 2002. Evaluating ontological decisions with ontoclean. *Communications of ACM* 45(2):61–65.

Guillen-Scholten, J.; Arbab, F.; de Boer, F.; and Bonsangue, M. 2003. A channel based coordination model for components. *ENTCS* 68(3).

Masolo, C.; Vieu, L.; Bottazzi, E.; Catenacci, C.; Ferrario, R.; Gangemi, A.; and Guarino, N. 2004. Social roles and their descriptions. In *Procs. of KR'04*, 267–277. AAAI Press.

Steimann, F., and Mayer, P. 2005. Patterns of interface-based programming. *Journal of Object Technology*.

Steimann, F. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering* 35:83–848.

Wong, R.; Chau, H.; and Lochovsky, F. 1997. A data model and semantics of objects with dynamic roles. In *Procs. of IEEE Data Engineering Conference*, 402–411.