Bridging Agent Theory and Object Orientation: Importing Social Roles in Object Oriented Languages

Matteo Baldoni[1], Guido Boella[1], and Leendert van der Torre[2]

[1]Dipartimento di Informatica. Università di Torino - Italy.
E-mail: {baldoni,guido}@di.unito.it
[2]CWI Amsterdam and Delft University of Technology. E-mail: torre@cwi.nl

Abstract. Social roles structure social institutions like organizations in Multi-Agent Systems (MAS). In this paper we describe how to introduce the notion of social role in programming languages. To avoid the commitment to a particular agent model, architecture or language, we decided to extend Java, the most prominent object oriented programming language, by adding social roles. The obtained language allows an easier implementation of MAS's w.r.t. the Java language. We also show that many important properties of social roles, studied in the MAS field, can be applied to objects. Two are the essential features of social roles according to an analysis reported in the paper: social roles are defined by other entities (called institutions), and when an agent plays a role it is endowed with powers by the institution that defines it. We interpret these two features into the object oriented paradigm as the fact that social roles are objects, which are defined in and exist only inside other objects (corresponding to institutions), and that, through a role, external objects playing the role can access to the object (institution) the role belongs to.

1 Introduction

Social roles are central in MAS since they are the basis for coordinating agents by means of organizations [1]. Roles are central also in object oriented modelling and programming (OO), where they are used to dynamically add behaviors to objects, to factorize features of objects like methods or access rights, and to separate the interactional properties of objects from their core behavior, thus achieving a separation of concerns.

Although it would surely be useful to find a unified notion of role, in both agent oriented (AO) and object oriented systems, the existence of many distinct notions of role (as well as of agent) makes this task a difficult challenge. Starting from the analysis of Boella and van der Torre [2], in this paper we describe how to introduce the notion of social role in programming languages. Since it is difficult to choose among the different agent systems and languages proposed by the MAS community, because each of them has its own idiosyncrasies (many types of agents are used, from reactive to cognitive ones; many architectures are used, from mobile to robotic ones; different definitions of organizations with social roles are used, from groups [4] to set of rules [5]), we propose an implementation that is set in the more traditional OO framework, whilst using the analysis developed in MAS research. More specifically, the research question of this paper is: How to extend Java by introducing the notion of social role? To answer this

question we first analyze the relevant properties of social roles and, then, we map them to programming constructs in the OO context.

The choice of the Java language is due to the fact that it is one of the prototypical OO programming languages; moreover, MAS systems are often implemented in Java and some agent programming languages are extensions of Java, e.g., see the Jade framework [6]. In this way we can directly use roles offered by our extension of Java when building MAS systems or extending agent programming languages.

Furthermore, we believe that to contribute to the success of the Autonomous Agents and Multiagent Systems research, the theories and concepts developed in this area should be applicable also to more traditional views. It is a challenge for the agent community to apply its concepts outside strictly agent based applications, and the object oriented paradigm is central in Computer Science. As suggested also by Juan and Sterling [7], before AO can be widely used in industry, its attractive theoretical properties must be first translated to simple, concrete constructs and mechanisms that are of similar granularity as objects.

The methodology that we use in this paper is to map the properties of social roles to roles in objects. To provide a semantics for the new programming language, called `powerJava`, we use a mapping to pure Java by means of a precompilation phase.

In Section 2 we discuss how social roles can fit the ontology of OO. In Section 3 we provide our definition of social roles and in Section 4 we map it to the OO domain. In Section 5 we introduce `powerJava` and in Section 7 we describe how it is translated to Java. Conclusions end the paper.

2 Social roles among objects

Why should it be useful for the OO paradigm to introduce a notion of social role, as developed in MAS? Even if the utility of roles is widely recognized in OO for organizing software programs, the diversity of conflicting approaches witnesses some difficulties, as the survey of Steimann [8] shows.

The success of the OO paradigm in many disciplines (KR, SE, DB, programming languages) is due also to the large conceptual modelling work behind it. The object orientation paradigm is inspired to the ontology used by humans to conceptualize material reality, in particular the fact that objects are composed of other objects, that they can be classified in classes, and that each class offers a different distinct behavior. These features find straightforward counterparts in programming languages. In particular, the abstraction and encapsulation principles, polymorphism, modularity and software reuse can be realized by means of the notion of object with its methods, and of class hierarchy.

The likely reason why the object oriented paradigm cannot accommodate easily the notion of role is that the notion of role does not belong to the fragment of ontology to which object orientation refers.

In this paper we extend the domain of the reference ontology of OO to the domain of social reality, which social roles belong to. The ontology of *social reality* represents the conceptual model of the social life of humans. Researches in this domain mostly stem from the agent oriented paradigm as a way to solve coordination problems among

agents in multiagent systems. But it is also an area of interest of ontological research, like in [9,10].

The notion of social role refers to the structure of social entities like institutions, organizations, normative systems, or even groups. These social entities are organized in roles [1,4,5]. Roles are usually considered as a means to distribute the responsibilities necessary for the functioning of the institution or organization. Moreover, roles allow the uncoupling of the features of the individuals from those of their roles. Finally, roles are used to define common interaction patterns, and embed information and capabilities needed to communication and coordination [11]. E.g., the roles of auctioneer and bidder are defined in an auction, each with their possible moves.

We call our extension of Java `powerJava`, since the powers given by institutions to roles are a key feature of roles in our model. An example is the role of director of a department: a buying order, signed by the agent playing the role of director, is considered as a commitment of the institution, that will pay for the delivered goods.

3 Properties of social roles

We consider as characteristic of roles two properties highlighted respectively in the knowledge representation area [10] and in the multiagent system area [12].

Definitional dependence: The definition of the role must be given inside the definition of the institution it belongs to. This property is related to the foundation property of roles [13]: a role instance is always associated with an instance of the institution it belongs to.

Powers: When an agent starts playing a role in an institution, it is empowered by the institution: the actions which it performs in its role "count as" [14] actions of the institution itself. This is possible only because of the definitional dependence: since the role is defined by the institution it is the institution itself which gives it some powers.

Institutions like groups, organizations, normative systems are not material entities, since they belong to the *social* reality, which exists only as a construction of human beings. According to the model of Boella and van der Torre [15,16], social entities can be modelled as agents, albeit of a special kind since they act in the world via the actions of other agents. In [2,12], also *roles* are considered as (description of) agents.

In this work, agents - like their players and institutions are - are modelled as objects, and, thus, by the previous observation, roles are modelled as objects too. In order to work at the level of objects we do not consider typical properties of agents like autonomy or proactiveness.

To understand these issues we propose a running example. Consider the role "student". A student is always a student of some school. Without the school the role does not exist anymore: e.g., if the school goes bankrupt, the actor (e.g. a person) of the role cannot be called a student anymore. The institution (the school) also specifies which are the properties of the student which extend the properties of the person playing the role of student: the school specifies the role's enrollment number, its email address in the school intranet, its scores at past examinations. Most importantly the school also

specifies how the student can behave. For example, the student can give an exam by submitting some written examination; this action is clearly defined by the school since it is the school which specifies how an examination is valued and it is the school which maintains the official records of the examinations which is updated with the new mark. Finally, the student can contact the secretary who is obliged to provide it with an enrollment certificate; also this action depends on the definition the school gives both to the student role and to the secretary role, otherwise the student could not have an effect on the her.

But in defining such actions the school *empowers* the person who is playing the role of student.

4 Modelling roles as objects

To translate the notion of social role in OO we need to find a suitable mapping between the agent domain and the object domain. The basic idea is that agents are mapped to objects. Their behaviors are mapped in methods invoked on the objects. We have to distinguish at least three different kinds of agents:

- Players of roles: their basic feature is that they can exercise the powers given by their roles when they act in a role, since their actions "count as" actions of their roles [14].
- Institutions: their basic feature is to have parts (roles) which are not independent, but which are defined by themselves. They must give to the defined roles access to their private fields and methods.
- Roles: they describe how the player of the role is connected to the institution via its powers. They do not exist without the institution defining them and they do not act without the agent playing the role.

The mapping between agents and objects must preserve this classification, so we need three kinds of objects.

- Objects playing roles: when they play a role, it is possible to invoke on them the methods representing the powers given by the role.
- Institutions: their definition must contain the definition they give to the roles belonging to them.
- Roles: they must specify which object can play the role and which powers are added to it. They must be connected both to the institution, since the powers have effect on it, and to the player of the role.

In OO terms, the player of the role can determine the behavior of the object, in which the role is defined, without having either a reference to it or access to its private fields and methods. In this way, it is possible to exogenously coordinate its behavior, as requested by Arbab [17].

In the next sections we will address in details the three different kinds of objects we need to model in `powerJava`.

4.1 Playing a role

An object has different (or additional) properties when it plays a certain role, and it can perform new activities, as specified by the role definition. Moreover, a role represents a specific state which is different from the player's one, which can evolve with time by invoking methods on the roles (or on other roles of the same institution, as we have seen in the running example). The relation between the object and the role must be transparent to the programmer: it is the object which has to maintain a reference to its roles. 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 role instance. A role is not an independent object, it is a facet of the player.

Since an agent can play multiple roles, the same method will have a different behavior, depending on the role which the object is playing when it is invoked. It must be sufficient to specify with is the role of a given object we are referring to. On the other hand, methods of a role can exhibit different behaviors according to whom is playing a role. So a method returning the name of the student together with the name of the school returns not only a different school name according to the school, but also a different value for the name according to whom is playing the role of student.

Note that roles are always roles in an institution. Hence an object can play at the same moment a role more than once, albeit in different institutions. For example, one can be a student at the high school, a student of foreign languages in another school, *etc*. We do not consider in this paper the case of an object playing the same role more than once in the same institution. However, an object can play several roles in the same institution. For example, 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 to, 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. Recall that to make this casting we do not only have to specify which role we are referring to, but also the institution where the object is playing the role, too. Otherwise, if an object plays the same role in more than one institution, the cast would be ambiguous.

We call this *role casting*. Type casting in Java allows to see the same object under different perspectives while maintaining the same structure and identity. In contrast, role casting conceals a *delegation* mechanism: the delegated object can only act as allowed by the powers of the role; it can access the state of the institution and, by exploiting a construct that will be introduced shortly (`that`) can also refer to the delegating object.

4.2 Institutions defining roles

The basic feature of institutions, as intended in our framework, is to define roles inside themselves. If roles are defined inside an institution, they can have access to the private variables and methods of the institution. The "definition" of an object must be read as the definition of the class the object is an instance of, thus, we have that the

class defining an institution includes the class definition of the roles belonging to the institution.

The fact that the role class definition is included inside the institution class definition determines some special properties of the methods that can be invoked on a role. In fact, for the notion of role to be meaningful, these methods should go beyond standard methods, whose implementation can access the private state of the role only. Roles add powers to objects playing the roles. Power means the capability of modifying also the *state of the institution* which defines the role and the *state of the other roles* defined in the same institution. This capability seems to violate the standard encapsulation principle, where the private variables and methods are visible only to the class they belong to: however, here, the role definition is itself inside the class definition, so encapsulation is not violated. This means also that the role must have a reference to the institution, in order to refer to its private or public methods and fields.

In our example, the method by which a student takes an examination must modify the private state of the school. If the exam is successful, the mark will be added to the registry of exams in the school. Similarly, if the method of asking the secretary a certificate should be able to access the private method of the secretary to print a certificate.

In MAS, roles can be played by different agents, it is sufficient that they have the suitable capabilities. This is translated in OO as the fact that to play a role an object must implement the suitable methods. In Java this corresponds to implementing an interface, i.e., a collection of method signatures. To specify who can play it, a role specifies an interface representing the requirements to play a role. Thus, an object to play a role must implements an interface.

The objects which can play the role can be of different classes, so that roles can be specified independently of the particular classes playing the role. This possibility is a form of polymorphism which allows to achieve a flexible external coordination and to make roles reusable.

At the same time a role expresses the powers which can be exercised by its player. Again, since powers are mapped into methods, a role is related to another interface definition. In summary, a role has two faces (see also Figure 1):

- It describes the methods that an object must show in order to play/enact the role. We call them *requirements*.
- It describes the methods that are offered by the role to an object that might enact it. We call them *powers*.

For Steimann and Mayer [18] roles define a certain behavior or protocol demanded in a context independently of how or by whom this behavior is to be delivered. In our model this translates to the fact that a role defines both the behavior required by the player of the role and the behavior offered by playing the role. However, the implementation of both the requested and offered behavior is not specified in the role.

The implementation of the requirements is obviously given inside the class of the object playing the role. The implementation of the powers must be necessarily given in the definition of the institution, which the role belongs to; the reason is that only in this way such methods can really be powers: they can have access to the state of the institution and change it.

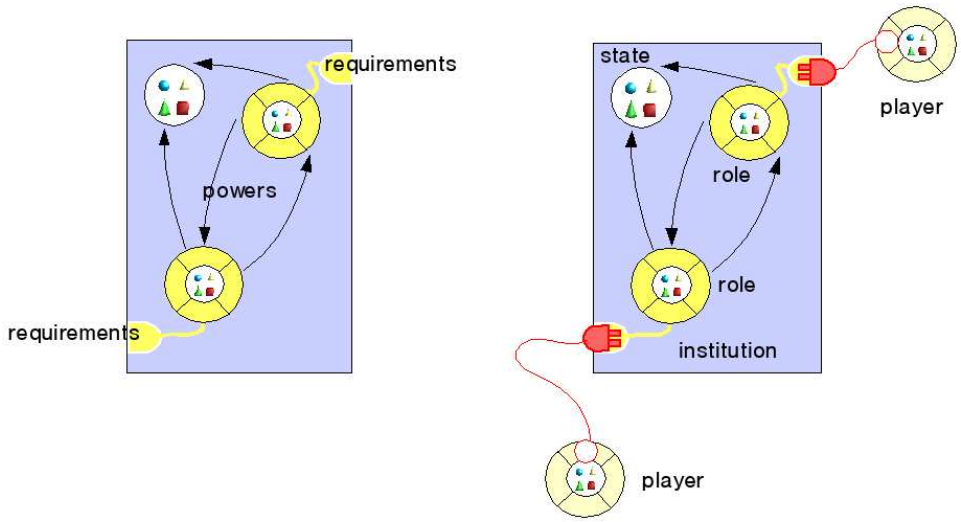

Fig. 1. The players will interact according to the acquired powers (they will follow the *protocol* implemented by the institution and its roles).

5 Introducing roles in Java: `powerJava`

We now have all the elements to introduce roles as the new construct in `powerJava`.

5.1 The syntax of `powerJava`

To introduce roles in `powerJava` we need very limited modifications of the Java syntax (see sketch in Figure 2):

1. A construct specifying the role with its name, requirements and powers (non-terminal symbol `rolespec`).
2. A construct that allows the implementation of a role, inside an institution and according to the specification of its powers (non-terminal symbol `roledef`).
3. A role casting construct, together with the specification of the institution to which the role belongs (non-terminal symbol `rcast`).

Note that nothing is required for an object to become the player of a role, apart from having the appropriate behavior required by the role specified by the keyword `enacts`.

The definition of a role using the keyword `role` is similar to the definition of an interface: it is in fact the specification of the powers acquired by the role in the form of methods signatures. The only difference is that the role specification refers also to another interface (e.g., `StudentRequirements` in Figure 3), that in turn gives the requirements to which an object, willing to play the role, must conform. This is implemented by the keyword `playedby`. This mechanism mirrors the idea, discussed in the previous section, that roles have two faces: the requirements and the powers. In the

```
rolespec := "role" identifier "playedby"
  identifier interfacebody

classdef ::= ["public"|"private"|...]
  "class" identifier ["enacts" identifier*]
  ["extends" identifier] ["implements" identifier*]
  classbody

classbody ::= "{" fielddef* constructors*
  methoddef* roledef* "}"

roledef ::= "definerole" identifier
  ["enacts" identifier*] rolebody

rolebody ::= "{" fielddef* methoddef* roledef* "}"

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

Fig. 2. Syntax

example, `role` specifies the powers of `Student`, whilst `StudentRequirements` - trivially - specifies its requirements.

Roles must be implemented inside an institution; the keyword `definerole` has been added to implement a role inside another class. A role implementation is like an inner-class definition. It is not possible, however, to define constructors; only the predefined one is available, having as a parameter the player of the role. Moreover, the definition of a role can contain other roles in turn (in this case the role itself becomes an institution). Finally, it is worth noting that the definition of institution is a class which can be extended by means of the normal Java constructs but the roles cannot be overridden.

Since the behavior of a role instance depends on the player of the role, in the method implementation the 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 used only in the role implementation. The value of `that` is initialized when the constructor of the role is invoked. Notice that the type of the referred object is the type defined by the role requirements or a subtype of it.

The greatest conceptual change in `powerJava` is the introduction of role casting expressions with the original Java syntax for casting. A `rcast` specifies both the role and the instance of the institution the role belongs to (or no object in case of a single institution). Note that the casting of an object returns an object which can be manipulated as any other object invoking methods and accessing variables on it.

We do not need a special expression for creating roles since we use the notation of Java for inner classes: starting from an institution instance the keyword “new” allows the creation of an instance of the role as if the role were an inner class of the institution. For example, let us suppose that `harvard` is a instance of `School` and that `chris` is

```
role Student playedby StudentRequirements {
  public String getName ();
  public void takeExam (int examCode, String text);
  public int getMark (int examCode);
}

interface StudentRequirements {
  public String getName ();
  public int getSocSecNum ();
}
```

Fig. 3. Specification of the powers and requirements.

a person who wants to become a student of `harvard`. This is expressed by the instruction `harvard.new Student(chris)`, using the predefined parameter having the role requirements `StudentRequirements` as type.

5.2 How to use `powerJava`

In Figures 3-5 we present our running example in `powerJava`. In Figure 3, the name of the role `Student` is introduced as well as the prototypes of the methods that constitute the powers and requirements. For example, returning the name of the `Student`, submitting a text as an examination, and so forth. As in an interface, no non-static variables can be declared. Differently from a Java interface, we couple a role with the specification of its requirements. This specification is given by means of the name of a Java interface, in this case, `StudentRequirements`, imposing the presence of methods `getName` and `getSocSecNum` (the person's social security number).

As explained, roles must be implemented inside some institution. In our running example (Figure 4), the role `Student` is implemented in a class `School`. The implementation must respect the method signature of the role powers. As for an inner class in Java, a role implementation has access to the private fields and methods of the outer class and of the other roles defined in the outer class; this possibility does not disrupt the encapsulation principle since all roles of an institutions are defined by who defines the institution itself. In other words, an object that has assumed a given role, by means of it, has access and can change the state of the corresponding institution and of the sibling roles. In this way, we achieve what envisaged by the analysis of the notion of role.

The object playing a role can be accessed by means of the special construct `that`, which refers to the object that enacts the role. In the example such an object has type `StudentRequirements`; the `that` construct is used in the method `getName()` in order to combine the player's name with the name of the school it attends. Like an instance of a class, a role instance can have a state, specified by its private fields, in this example, `studentID`.

In order for an object to play a role it is sufficient that it conforms to the role requirements. Since the role requirements are implemented as a Java interface, it is

```
class School {
  private int[][] marks;
  private String schoolName;
  public School (String name) {
    this.schoolName = name;
  }
  definerole Student {
    private int studentID;
    public void setStudentID (int studentID) {
      this.studentID = studentID;
    }
    public int getStudentID () {
      return studentID;
    }
    public void takeExam (int examCode, String text) {
      marks[studentID][examCode] = eval(text);
    }
    public int getMark (int examCode) {
      return mark[studentID][examCode];
    }
    public int getName () {
      return that.getName() + " at " + schoolName;
    }
  }
  public int eval (String text){...}
}
```

Fig. 4. Defining the institution and implementing a role specification.

sufficient that the class of the object implements the methods of such an interface. In Figure 4, the class `Person` can play the role `Student`, because it conforms to the interface `StudentRequirements` by implementing the methods `getName` and `getSocSecNum`.

A role is created by means of the construct `new` as well as it is done in Java for inner class instance creation. For example, (see Figure 5, method `main` of class `TestRole`), the object referred by `chris` can play the part of the student of the school `harvard` by executing the following instruction: `harvard.new Student(chris)`. In this context, i.e. within the role definition, `that` will refer to `chris`. Moreover, note that the same person can play the same role in more than one school. In the example `chris` is also a student of `mit`: `mit.new Student(chris)`.

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 referent to its player object. Methods can be invoked from the players of the role, given that the player is seen in its role (e.g. `Student`). This is done in `powerJava` by casting the player of the role to the role we want to refer to.

```
class Person enacts Student {
  private String name;
  private int SSNumber;
  public Person (String name) { this.name = name; }
  public String getName () { return name; }
  public int getSocSecNum () { return SSnumber; }
}

class TestRole {
  public static void main(String[] args) {
    Person  chris = new Person("Christine");
    School  harvard = new School("Harvard");
    School  mit = new School("MIT");
    harvard.new Student(chris);
    mit.new Student(chris);
    String x=((harvard.Student)chris).getName();
    String y=((mit.Student)chris).getName();
  }
}
```

Fig. 5. Palying a role.

We use the Java cast syntax with a difference: the object is not casted to a type, but to a *role*. However, since roles do not exist out of the institution defi ning 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 `rcast` (see Figure 2). For instance, `((harvard.Student) chris).getName()` takes `chris` in the role of student in the institution `harvard`. As a result, if `getName` applied to `chris` initially returned only the person's name, after the cast, the same invocation will return "Christine at Harvard". Obviously, if we cast `chris` to the role of student at `mit` `((mit.Student) chris).getName()`, we obtain "Christine at MIT".

With respect to type casting, role casting does not only selects the methods available for the object, but it changes also the state of the object and the meaning of the methods: here, the name returned by the role is different from the name of the player since the method has a different behavior. As it is done in Java for the interfaces, roles 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.* [19]'s principle of "programming to an interface".

`powerJava` allows the defi nition of roles which can be further articulated into other roles. For example, a school can be articulated in school classes (another social entity) which are, in turn, articulated into student roles. This is possible because, as we discuss in next section, roles are implemented by means of classes, which can be nested one into the other. In this way, it is possible to create a hierarchy of social entities, where each entity defi nes the social entities it contains. As described by [12], this hierarchy recalls the composition hierarchy of objects, which have other objects as their parts.

6 An example about protocols

Hereafter, we report an example set in the framework of interaction protocols, describing an implementation of well-known *contract net* protocol [3] in our language. Contract net is used in electronic commerce and in robotics for allowing object of the class `Agent` which are unable to do some task to have them done. The protocol is only concerned with the realization of a specific pattern of interaction, in which the manager sends a call for proposal to a set of bidders. Each bidder can either accept and send a proposal or refuse. The manager collects all the proposals and selects one of them.

The `powerJava` implementation comprises the roles of `Manager` and that of `Bidder`. A `Manager` has the power of of starting a negotiation. `Bidders` have the power of taking part to a negotiation. The contract net protocol is the institution inside which the two roles are defined. Notice that the capability of the `Bidder` of defining a proposal as well as that of the `Manager` of evaluating the proposals depend on the specific task that is the object of the negotiation and on the business logics of the two role players. The requirements of the two roles express the need of having this capabilities in the role players.

```
role Manager {
  public void startNegotiation(Task task);
}
interface ManagerReq {
  public int evaluateProposal(Proposal[] proposal);
  public void receiveResult(Object result);
}

interface Bidder {
  public void partecipateNegotiation();
}
interface BidderReq {
  public boolean evaluateTask(Task task);
  public Proposal getProposal(Task task);
  public void removeProposal(Task task, Proposal proposal);
  public ResultTask performTask(Task task);
}

class ContractNetProtocol {
  Task task;
  Manager manager;
  Bidders[] bidders;
  Proposal[] proposals;
  int i; int count;
  public ContractNetProtocol() {
      // initializes the state
  }

  definerole Manager {
    public void startNegotiation(Task task) {
      ContractNetProtocol.this.manager = that;
      ContractNetProrocol.this.task = task;
```

```
    for (int i=0; i < count; i++)
      bidders[i].cfp(task);
    }
    private void refuse(Bidder bidder) {
      i = i + 1;
      if (i >= count) notifyBidders();
    }
    private void propose(Proposal proposal, Bidder bidder) {
      i = i + 1;
      proposals[bidder.getID()] = proposal;
      if (i >= count) notifyBidders();
    }
    private void failure(TaskExecException err, Bidder bidder) {
      that.receiveResult(err);
    }
    private void inform(ResultTask result, Bidder bidder) {
      that.receiveResult(result);
    }
    private void notifyBidders() (
      int selectedProposal =
        that.evaluateProposals(proposals);
      bidders[selectedProposal].acceptProposal(
          proposals[selectedProposal]);
        for (int j=0; j<count; J++)
          if (selectedProposals != j)
            bidders[j].refuseProposal(
                proposals[selectedProposal]);
    }
}
definerole Bidder {
  int ID;
  public void partecipateNegotiation() {
    // add this new bidder to the array of bidders
    // assign an ID and increments count
  }
  private void cfp() {
    if (that.evaluateTask(task))
      manager.propose(that.getProposal(task));
    else
      manager.refuse(this);
  }
  private void refusePoposal(Proposal proposal) {
    that.removeProposal(proposal);
  }
  private void acceptProposal(Proposal proposal) {
    try {
      manager.inform(that.performTask(proposal, task)), this);
    } catch(TaskExecException err) {
      manager.failure(err, this);
    }
```

```
    }
  }
}
```

Notice that in LifeTimeManager, which is the part of the code in which three "agents" are created and used to play a Manager and two Bidders, to carry on the negotiation it is sufficient that the players respectively invoke the power for initiating and the power for partecipating to the negation itself. The interaction at this level is "hidden" because it is carried on within the institution corresponding to the protocol. For the sake of simplicity the code does not contain references to threads, which are indeed necessary for a correct execution. An object of class Agent that shows a complete set of requirements could play different roles even at the same time even in the same instance of protocol.

```
class LifeTimeManager {
  public static void main(String[] args) {
    Agent initiator = new Agent(...);
    Agent partecipant1 = new Agent(...);
    Agent partecipant2 = new Agent(...);
    ContractNetProtocol cnp = new ContractNetProtocol();
    cnp.new Manager(initiator, task);
    cnp.new Bidder(partecipant1);
    ((cnp.Bidder)partecipant1).partecipateNegotiation();
    cnp.new Bidder(partecipant2);
    ((cnp.Bidder)partecipant1).partecipateNegotiation();
    ((cnp.Manager)initiator).startNegotiation();
  }
}
```

7 Translating roles in Java

In this section we provide a 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, as, e.g., [17] proposes for introducing components and channels in Java, or in the way inner classes are implemented in Java. The precompiler has been implemented by means of the tool javaCC, provided by Sun Microsystems [20].

The role definition is simply an interface (see Figure 6) 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 to constrain the creation of role instances relatively to players that conform to the requirements.

While a role definition is precompiled into a Java interface, a specific role implementation is precompiled into a Java inner class which implements such an interface. The inner class resides in the class that implements the institution. For example, the implementation of the role `Student` in the class `School` is precompiled into an inner class of `School`, named automatically `StudentPower`. `StudentPower` implements the interface into which the role is translated, `Student`. The `that` construct,

```
interface Student {
  public String getName();
  public void giveExam(int examCode, String text);
  public int getMark(int examCode);
}

class Person enacts StudentRequirements {
  private java.util.Hashtable studentList =
    new java.util.Hashtable();
  public void setStudent (Student sp, Object inst) {
    studentList.put(inst, sp);
  }
  public Student getStudent (Object inst) {
    return studentList.get(inst);
  }
  private String name;
  private int SSNumber;
  public Person (String name) { this.name = name; }
  public String getName() { return name; }
  public int getSocSecNum () { return SSNumber; }
}
```

Fig. 6. Translation of a role and its player.

which keeps the relation between the player instance and the role instance, is precompiled into a field of `StudentPower` of type `StudentRequirements`. This field is automatically initialized by means of an *ad hoc* constructor `School`. This predefined constructor is introduced by the precompiler in the inner class and it takes the player as a parameter which must have the type required by the role definition. In this case `StudentRequirements`.

All the constructor does is to initialize the `that` parameter with the player instance and to manipulate the player instance in order to let it have a referent to the role instance. This is necessary for establishing a correspondence between the instance of the player class and the instance of the inner class. The remaining link between the instance of the inner class and the outer class defining it (the institution) is provided automatically by Java (e.g., `School.this`).

Since every object can play many roles simultaneously, it is necessary to keep, related to the object at hand, the set of its roles. This is obtained by adding, at precompilation time, to every class for each different kind of role that it can play, a structure for book-keeping its role instances. As an example, `Person` enacts the role `Student`. So its instances will have a hash-table that keeps the many student roles played by them in different institutions. In the case of `chris` there will be an instance corresponding to the fact that she is a student of `harvard` and one for her being a student of `mit`. Methods for accessing to this structure are supplied. In the example they allow setting and getting the Student role: `setStudent` and `getStudent`. Notice that book-keeping

```
class School {
  public School (String schoolName) {
    this.schoolName = schoolName;
  }
  class StudentPower implements Student {
    StudentRequirements that;
    public Student (StudentRequirements that) {
      this.that = that;
      (this.that).setStudent(this, School.this);
    //role's fields and methods ...
  }
  //institution's fields and methods ...
}
```

Fig. 7. Translation of institution.

```
class TestRole {
  public static void main(String[] args) {
    Person  chris = new Person("Christine");
    School  harvard = new School("harvard");
    School  mit = new School("MIT");
    harvard.new StudentPower(chris);
    mit.new StudentPower(chris);
    String x = chris.getStudent(harvard).getName();
    String y = chris.getStudent(mit).getName();
  }
}
```

Fig. 8. Translation of main.

could be implemented in a more general way, using just one hash table and indexing w.r.t. the institution and the role.

Finally, we describe how role casting is precompiled. The expression referring to an object in its role (a `Person` as a `Student`, e.g., `(harvard.Student)chris`) is translated into the selector returning the reference to the inner class instance, representing the desired role w.r.t. the specified institution. The translation will be `chris.getStudent(harvard)` (see Figure 7).

A summary of all this translation is shown in Figure 9 as an UML class diagram, where dashed lines represent the newly introduced concepts.

8 Conclusion

In this paper, we extend Java by introducing the notion of social role developed in MAS. The basic features of roles in our model are that they are definitionally dependent on the institution they belong to, and they offer powers to the entities playing them. We map

agents, institutions and roles to objects, and powers to methods, that are offered by roles to the objects playing those roles. The characteristic feature of powers is that they can access the private fields and methods of the institution they belong to and those of the sibling roles defined in the same institution. In order to allow an object to be seen in the role it plays we extend the notion of casting offered by Java: type casting in Java allows to see the same object under different perspectives while maintaining the same structure and identity; in contrast, role casting allows to see the object as having a different state and different methods, as specified by the role powers.

We are currently working at an extension of `powerJava` some preliminary results can be found in [24]. In particular, in this work `powerJava` is compared to proposals coming from the Object-Oriented community.

Our approach shares the idea of gathering roles inside wider entities with languages like Object Teams [21] and Ceasar [22]. These languages emerge as refinements of aspect oriented languages aiming at resolving practical limitations of other languages. In contrast, our language starts from a conceptual modelling of roles and then it implements the model as language constructs. Differently than these languages we do not model aspects. The motivation is that we want to stick as much as possible to the Java language. However, aspects can be included in our conceptual model as well, under the idea that actions of an agent playing a role "count as" actions executed by the role itself. In the same way, the execution of methods of an object can give raise by advice weaving to the execution of a method of a role. On the other hand, these languages do not provide the notion of role casting we introduce in `powerJava`. Roles as double face interfaces have some similarities with Traits [23] and Mixins. However, they are distinguished because roles are used to extend instances and not classes.

By implementing roles in an OO programming language, we gain in simplicity in the language development, importing concepts that have been developed by the agent community inside the Java language itself. This language is, undoubtedly, one of the most successful currently existing programming languages, which is also used to implement agents even though it does not supply specific features for doing it. The language extension that we propose is a step towards the overcoming of these limits.

At the same time, introducing theoretically attractive agent concepts in a widely used language can contribute to the success of the Autonomous Agents and Multiagent Systems research in other fields. Developers not interested in the complexity of agent systems can anyway benefit from the advances in this area by using simple and concrete constructs in a traditional programming language.

Future work concerns, on one hand, the provision of a formal semantics to `powerJava` and the extension of the Java type system with roles; on the other hand, the role construct of `powerJava` can be extended, for example, by allowing roles playing roles (e.g., a student can play the role of representative in the school), and we also study how our definition of social roles can directly be used in Java based agent programming languages, in frameworks like Jade [6].

In this paper we present a "lite" version of the `powerJava` language. We are currently developing a full fledged version that allows more natural programming for the Java expert, in which the role implementation does not require a specific construct (`definerole`), but it entirely relies upon the inner class definition mechanism. Such

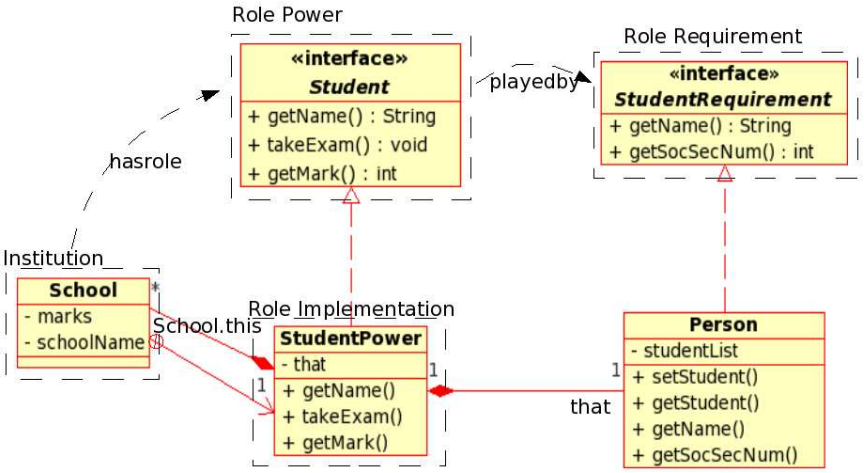

Fig. 9. The UML class diagram.

inner classes must implement the role specifi cations. The advantages are many: on a hand, one can have more implementations of a role inside the same institution, inner classes can enact other roles, they can be institutions themselves, and use extensions.

References

1. Bauer, B., Muller, J., Odell, J.: Agent UML: A formalism for specifying multiagent software systems. Int. Journal of Software Engineering and Knowledge Engineering **11(3)** (2001) 207–230
2. Boella, G., van der Torre, L.: Attributing mental attitudes to roles: The agent metaphor applied to organizational design. In: Procs. of ICEC'04, IEEE Press (2004)
3. Davis, R., and Smith, R. G.: Negotiation as a metaphor for distributed problem-solving. In Artificial Intelligence, 20, 1983.
4. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: an organizational view of multiagent systems. In: LNCS n. 2935: Procs. of AOSE'03, Springer Verlag (2003) 214–230
5. Zambonelli, F., Jennings, N., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. IEEE Transactions of Software Engineering and Methodology **12(3)** (2003) 317–370
6. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with a FIPA-compliant agent framework. (Software - Practice And Experience) 103–128
7. Juan, T., Sterling, L.: Achieving dynamic interfaces with agents concepts. In: Procs. of AAMAS'04. (2004)
8. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering **35** (2000) 83–848
9. Boella, G., van der Torre, L.: An agent oriented ontology of social reality. In: Procs. of FOIS'04, Torino (2004) 199–209
10. Masolo, C., Vieu, L., Bottazzi, E., Catenacci, C., Ferrario, R., Gangemi, A., Guarino, N.: Social roles and their descriptions. In: Procs. of KR'04. (2004)

11. Cabri, G., Ferrari, L., Leonardi, L.: Agent role-based collaboration and coordination: a survey about existing approaches. In: IEEE Systems, Man and Cybernetics Conference. (2004)
12. Boella, G., van der Torre, L.: Organizations as socially constructed agents in the agent oriented paradigm. In: Procs. of ESAW'04, Berlin, Springer Verlag (2004)
13. Guarino, N., Welty, C.: Evaluating ontological decisions with ontoclean. Communications of ACM **45(2)** (2002) 61–65
14. Searle, J.: The Construction of Social Reality. The Free Press, New York (1995)
15. Boella, G., van der Torre, L.: Groups as agents with mental attitudes. In: Procs. of AAMAS'04, ACM Press (2004) 964–971
16. Boella, G., van der Torre, L.: Regulative and constitutive norms in normative multiagent systems. In: Procs. of KR'04, AAAI Press (2004) 255–265
17. Arbab, F.: Abstract behavior types: A foundation model for components and their composition. In: Formal Methods for Components and Objects, LNCS 2852. Springer Verlag, Berlin (2003) 33–70
18. Steimann, F., Mayer, P.: Patterns of interface-based programming. Journal of Object Technology (2005)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Software. Addison-Wesley (1995)
20. Java compiler compiler [tm] (javaCC [tm]) - the java parser generator. (Sun Microsystems) https://javacc.dev.java.net/.
21. Herrmann, S.: Object teams: Improving modularity for crosscutting collaborations. In: Procs. of Net.ObjectDays. (2002)
22. Mezini, M., K.Ostermann: Conquering aspects with caesar. In: Procs. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press (2004) 90–100
23. N. Scharli, S. Ducasse, O.N., Black, A.: Traits: Composable units of behavior. In Verlag, S., ed.: LNCS, vol. 2743: Procs. of ECOOP'03, Berlin (2003) 248–274
24. Baldoni, M., Boella, G., and van der Torre, L.: `powerJava` : Ontologically Founded Roles in Object Oriented Programming Languages. In D. Ancona and M. Viroli, editors, Proc. of 21st ACM Symposium on Applied Computing, SAC 2006, Special Track on Object-Oriented Programming Languages and Systems (OOPS 2006), Dijon, France, April 2006. ACM. To appear.