JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 2, No. 2, March–Aprile 2003

Interaction between Objects in powerJava

Matteo Baldoni, Università degli Studi di Torino - Italy
Guido Boella, Università degli Studi di Torino - Italy
Leendert van der Torre, University of Luxembourg - Luxembourg

In this paper we start from the consideration that high level interaction between entities like web services has very different properties with respect to the interaction between objects at the lower level of programming languages in the object oriented paradigm. In particular, web services, for security, usability and user adaptability reasons, offer different operations to different users by means of access control and keep track of the state of the interaction with each user by means of sessions. The current vision in object orientation, instead, considers attributes and operations of objects as being objective and independent from the interaction with another object, which is sessionless. To introduce these features in the interaction between objects directly in object oriented programming languages, we take inspiration from how access control is regulated by means of roles. Roles allow objects to offer different operations depending on the type of the role, of the type and identity of the player of the role, and to define session-aware interaction.
We start from a definition of roles given in ontologies and knowledge representation and we discuss how this definition of roles can be introduced in Java, building our language powerJava.

1 INTRODUCTION

Object orientation is a leading paradigm in programming languages, knowledge representation, modelling and, more recently, also in databases. The basic idea of object orientation is that the attributes and operations of an object should be associated with it. The interaction with the object is made possible by the public attributes specified by the class which the object is an instance of and by its public operations. The implementation of an operation is encapsulated in the class of the object and can access its private state. This structure allows programs to fulfill the data abstraction principle: the public attributes and operations are the only possibility to manipulate an object and their implementation is not visible to the other objects manipulating it; thus, the implementation of the class can be changed without changing the interaction capabilities of its instances.

This view can be likened with the way we interact with objects in the world: the same operation of switching a device on by pressing a button is implemented in different manners inside different kinds of devices, depending on their functioning.

However, in computer science, other kinds of interaction between entities have been devised at levels higher than programming languages. In particular, the interaction at the level of web services has different properties, in order to satisfy security, usability and user adaptation requirements: different operations are offered to different users, the execution of an operation depends on the identity of the caller of the operation, and the state of the interaction with a user is maintained in a *session*.

Albeit these features can be introduced by programming, e.g., by means of patterns, the lack of these abstractions at the lower level limits sometimes the potentialities of object oriented languages:

1. Despite the method invocation mechanism is based on the metaphor of sending messages to objects, there is no notion of the sender of a message. Thus, the *caller* object (e.g., the `this` in Java) invoking a method of another object (the *callee*) is not taken into account for the method execution. Hence, when an operation is invoked its meaning can not depend on the caller's identity and class.

2. All caller objects of whatever classes can access all the public attributes and invoke all the public operations of every other callee object. Hence, it is not possible to distinguish which attributes and operations are visible for which classes of caller objects.

3. The callee object can exhibit a single interface to all the callers, and methods can have only one implementation in the callee.

4. The values of the private and public attributes of a callee object are the same for all other caller objects. Hence, the callee object exhibits only one state.

5. The interaction with a callee object is *sessionless* since the invocation of an operation does not depend on the identity of the caller (1) and there is only one state (4). Hence, the value of attributes and, consequently, the meaning of operations cannot depend on the previous interactions between the callee and each caller object.

6. Finally, the operational interface of abstract data types induces an asymmetrical semantic dependency of the callers of operations on the operation provider: the caller takes the decision on what operation to perform, passes the values of the parameters, and then it relies on the provider to carry out the operation, without further interaction.

The limitations 2-4 hinder modularity, since it would be useful to keep distinct the core behavior of an object from the different interaction possibilities which it offers to different kinds of objects. Some programming languages offer ways to give multiple implementations of interfaces, but the dependance from the caller cannot be taken into account, unless the caller is explicitly passed as a parameter in all methods. The limitation 5 complicates the modelling of distributed scenarios where communication is based on protocols and sessions are required. The first and last ones complicate coordination of components: method invocation does not allow objects to reach a minimum level of “control from the outside” of the participating objects [3].

The reason of these problems rests in the philosophy behind object orientation, which is based on the ontological assumption that attributes and operations of objects are objective: they are the same whatever is the caller object, unless it is passed as an explicit parameter. To solve this problem at the level of programming constructs we take inspiration from the solution used to control access to the services of a system. For security reasons, it is necessary to make explicit the notion of caller of an operation. In particular, in the role based access control model (RBAC) [38], access rights are associated with roles and users - the callers of operations - are made members of appropriate roles, thereby acquiring the roles’ permissions. Moreover, sessions are mappings between a user and an activated subset of roles that are assigned to the user.

Like in the RBAC model, in our paper roles are based on an *organizational metaphor*: roles describe the way persons can interact with and within an organization, assigning them institutional powers. The particular way of interaction with an organization depends on the properties of the person it is interacting with and to what the organization allows him to do in the role he plays. So, an organization does not offer only a direct single way of interacting with it, but it is possible to interact with the organization only via roles, where roles are defined by the organization itself.

Instead, most other works on roles in programming languages adopt a different perspective: roles are seen as a way to extend the behavior of an object (e.g., [16, 21, 23, 26, 37]) and not as a way to model how an object offers different possibilities of interaction to different kind of players and maintains the session of interaction. Thus, we pass from a *player-centered* vision of roles to an *organizational-centered* one.

Our definition of roles emerges from the analyses of organizational roles made in ontologies and knowledge representation [13, 15, 28, 29]. Thus, we not only introduce roles in object oriented programming languages motivated by practical considerations, as discussed above, but we also introduce a notion of role which is well founded and on which there is wide agreement among authors in ontologies and knowledge representation.

The methodology we use is to introduce roles in a real programming language, Java, one of the most used object oriented languages. To prove the feasibility of this approach, and give a semantics, we translate the new language, called *powerJava*, to pure Java by means of a pre-compilation phase.

The structure of the paper is as follows. First we introduce our ontological definition of roles taking inspiration from a running example. Then we present the powerJava language and its translation. Related work and summary close the paper.

2 THE INTERACTION WITH OBJECTS VIA ROLES

Despite several proposals have been made about how to introduce roles in programming languages, there is little consensus on which are their properties and their purpose. A possible reason for this divergence is that the notion of role is a commonsense one, and, thus, it is fuzzy: everyone has an intuitive understanding of this notion, but partially different from the others.

To avoid resorting to intuition only, in this paper we introduce roles in object orientation starting from the analyses performed in the field of ontology and knowledge representation, also by the authors of this paper [13, 15], so to have a precise definition of roles on which there is widespread agreement and which is justified independently from the practical problems we want to solve with it. This definition of roles uses a metaphor taken directly from organizational management. Organizations, and more generally institutions, are not like standard objects which can be manipulated from the outside (e.g., a radio can be switched on). Rather, institutions are objects belonging to the social reality, and the interaction with them can be performed only via the roles which they offer [14].

Roles are useful not only to model domains that include institutions and organizations. Rather, every object can be considered as an institution or an organization structured in roles, if it is necessary to model in different ways the interaction of this callee object with different types of caller objects, depending on their class and on their aims, or to keep track of the interaction with each caller object.

To make an example, let us suppose to model a class `Printer`. The interaction possibilities offered by the class are different and depend on which objects invoke its methods. For example, some objects have more privileges than other ones, and thus they can invoke methods which are not available to other objects interacting with the same printer. Moreover, some methods keep track of the interaction with each specific object invoking them. For example, `print` counts the number of pages printed by each object invoking it to check whether the quota assigned to the object is respected. However, objects with more privileges do not have a quota of printed pages.

The `Printer` can be seen as an institution which supplies two different roles for interacting with it (the set of methods a caller can invoke): one role of normal `User`, and the other role of `SuperUser`. The two roles offer some common methods (roles are classes) with different implementations, but they also offer other different methods to their players (and there is no direct way to interact with the `Printer`). For example, `User`s can print their jobs and the number of printable pages is limited to a given maximum; thus, the number of pages is counted (the role associates new attributes with the player): each `User` should be associated with a different state of the interaction (the role has an instance with a state which is associated with its player). The `User` can `print` since the implementation of its methods has access to the private methods of the `Printer` (the methods of the `User` access the private attributes and operations of another object, the institution). `SuperUsers` have the method `print` with the same signature, but with a different implementation: they can print any number of pages; moreover, they can reset the page counter of `Users` (a role can access the state of another role, and, thus, roles coordinate the interaction).

A role like `SuperUser` can access the state of the other `User` roles and of the callee object (the institution `Printer`) in a safe way only if it encapsulated in the institution `Printer`. Thus the definition of the role must be given by the same programmer who defines the institution (the class of the role belongs to the same namespace of class of the institution, or, in Java terminology, it is included in it).

In order to interact as `User` or `SuperUser` it is necessary to exhibit some requested behavior. For example, in order to be a `User` a caller object must have an account (it must be a `Accounted`), which is printed on the pages (returned by a method offered by the player of the role). A `SuperUser` can have more demanding requirements.

Finally, a role `User` can be played only when there is an instance of `Printer` and an instance of a class implementing `Accounted` which can play the role.

This example highlights the following properties that organizational roles have in our ontological model we discuss in [13, 15]:

- *Foundation*: a (instance of) role must always be associated with an instance of the institution it belongs to, besides being associated with an instance of its player (extending Guarino and Welty [22]).
- *Definitional dependence*: the definition of the role must be given inside the definition of the institution it belongs to.
- *Institutional empowerment*: the operations defined for the role in the definition of the institution have access to the attributes and operations of the institution and of the other roles: thus, we call them *powers*. Instead, the operations that a class must offer for playing a role are called *requirements*.

These features are considered also by other authors in ontologies and knowledge representation, as discussed in Section 5.

Contrary to natural classes like person, roles lack *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, a role can be played by different kinds of players. For example, the role of customer can be played by instances of person and of organization, two classes which do not have a common superclass. The role must specify how to deal with the different properties of the possible players. This requirement is in line with UML, which relates roles and interfaces as partial descriptions of behavior.

Hence, we propose quite a general definition of roles, independently from programming languages. But the example above illustrates how these features can be mapped on an object oriented scenario to solve the problems described in Section 1, as we discuss in the next section.

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

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

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

roleimpl  ::= "definerole" identifier rolebody

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

rcast     ::= (expr.identifier) expr

keyword   ::= that | ...
```

Figure 1: The extension of the Java (1.4) syntax in powerJava.

3 INTRODUCING ROLES IN JAVA: POWERJAVA

In this section we introduce our extension of Java by following three steps. First of all, roles are classes with private and public attributes and methods, but we will also describe how to specify them in an abstract way in terms of the signatures of the "requirements" and "powers", independently from their implementation in an institution. We call this description a role specification. A role specification is like a double-faced interface, reflecting the two aspects of the role. Second, we will introduce the way in which a role is implemented by a class in an institution with a particular attention to the definitional dependence of a role with respect to the institution within which it exists. We will do this inspired by Java inner classes. Last, we will show how an object can enter a role and, by playing the role, it can exercise its powers. The syntax of powerJava is illustrated in Figure 1.

Specification of powers and requirements

A role should be specified, for the sake of modularity, independently from its possible implementations as classes in specific institutions. To promote the view "program to an interface, not to an implementation" [19], we introduce role abstract specifications which must be respected by their implementations. Moreover, for Steimann and Mayer [41], roles define a certain behavior (or protocol) demanded in a context independently of how or by whom it is to be delivered. Thus, in order to make role systems reusable, it is necessary that a role can be played by more than one class only.

Specifying a role implies specifying both what is required to a caller in order to play it, and which powers the player acquires in the institution in which the role is implemented. Thus, a role specification has a list of abstract signatures of the methods offered to objects playing the role (*powers*) and a list of abstract signatures of the methods required to be implemented by the objects in order to be able to play the role (*requirements*). The latter ones are modelled as an interface associated with the `role` construct by the keyword `playedby`.

In this way, objects offering a role and objects which can play it can be developed independently of each other. In particular, any class implementing the requirement interface can play the role.

Let us consider again the `Printer` of the previous section: a normal `User` acquires the powers to `print` and to know the number of printed pages `getPrintedPages`, and it is required to provide its login (`getLogin`). The role specification for the user is the following, where the keyword `playedby` associates with an interface body specifying the powers the requirements specified by the separate interface `Accounted`:

```
role User playedby Accounted {
  int print(Job job);
  int getPrintedPages();
}

interface Accounted {
  Login getLogin();
}
```

The `SuperUser`, instead, must have both an account and a certificate to guarantee its profile, and it has additional powers:

```
role SuperUser playedby Certified {
  int print(Job job);
  int getTotalPrintedPages();
  void resetPrinterCounter(User user);
 }

interface Certified extends Accounted {
  Certificate getCertificate();
}
```

Institutions and role implementation

As discussed in Section 2, roles are always associated with an instance of an institution, and are definitionally dependent on it. We call the methods offered by roles "powers" because they offer the possibility to modify the private state and access private methods of the institution which defines them and the state of the other roles defined in the same institution. In this way roles allow a player to interact with(in) an institution. In our running example, the method `print`, both of a `User` and a `SuperUser`, will access a private counter of the printer for updating the total number of printed pages. The method `resetPrinterCounter` of `SuperUser`, instead, allows a player of the role `SuperUser` to change the state of a different role. Powers, thus, are methods which seem to violate the standard encapsulation principle, where the private variables are visible only to the class they belong to. However, here, the encapsulation principle is preserved by the definitional dependence property: the definition of all the roles of an institution depends on the definition of the institution; so it is the institution itself which gives to the roles access to its private fields and methods. Since it is the programmer of institution itself which implements its roles, there is no risk of abuse by part of the role of its access possibilities. Enabling a class to belong to the namespace of another class without requiring it to be defined as "friend", and thus endangering modularity, is achieved in Java by means of the *inner class* construct. The construct `definerole` allows the programmer to define a sort of inner class in order to implement a role specification inside an institution (the outer class). For example, the code in Figure 2 defines the class `Printer`, which contains the implementations of the above mentioned `User` and `SuperUser` role. The name of the class of these role implementation is respectively `Printer.User` and `Printer.SuperUser`. Note that, role specifications cannot be implemented in different ways in the same institution and we do not consider the possibility of extending role implementations (possible with inner classes). Even if in a preliminary version [8] this possibility has been considered, we omit it here because it would introduce some problems in requirement handling. If a role extends another role, the most specific inherits also the requirements of the other but, as we will better see in the next section, our translation cannot handle multiple requirements.

In order for an object to play a role it is necessary 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, e.g.:

```
class AccountedPerson implements Accounted {
  Login login;  // ...
  Login getLogin() {
    return login;
  }
}
```

A `CertifiedPerson` is defined in a similar way by implementing the `Certified` interface. Note that also other classes can implement the requirement interface and thus play the roles.

Since roles are classes which can be instantiated and the behavior of a role instance depends on its player, in the role method implementation, the player instance can be retrieved via a new reserved keyword: `that`. So this keyword refers to *that*

```
class Printer {
  final static int MAX_PAGES_PER_USER;
  private int totalPrintedPages = 0;

  private void print(Job job, Login login) {
    totalPrintedPages += job.getNumberPages();
    // performs printing
  }
  private boolean validCertificate(Certificate cert) {
    // checks the certificate cert
  }
  definerole User {    //  implementation of the role User
    private int counter = 0;
    public int print(Job job) {
      if (counter > MAX_PAGES_USER)
        throws new IllegalPrintException();
      counter += job.getNumberPages();
      Printer.this.print(job, that.getLogin());
      return counter;
    }
    public int getPrintedPages(){
      return counter;
    }
  }
  definerole SuperUser {    //  implementation of the role SuperUser
    public SuperUser() {
      //first, verify the identity of the player
      if (!validCertificate(that.getCertificate()))
        throw new Exception("You are not allowed to enter this role");
    }
    public int print(Job job) {
      Printer.this.print(job, that.getLogin());
      totalPrintedPages += job.getNumebrPages();
      return totalPrintedPages;
    }
    public int getTotalPrintedpages() {
      return totalPrintedPages;
    }
    public void resetPrinterCounter(User user) {
      ((Printer.User)user).counter = 0;
    }
  }
}
```

Figure 2: The class Printer with its User and SuperUser roles.

object which is playing the role at issue, and it is used only in the role implementation. An example is the invocation of `that.getLogin()` as a parameter of the method `print` in Figure 2. 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 given by the `playedby` keyword in the role specification. The fact of having two links, one to the player (`that`) and one to the institution (`Printer.this`), is actually an *invariant* of every role in our extension of Java.

For creating instances of the inner classes implementing roles, we use the Java inner class syntax: starting from an institution instance the keyword `new` allows the creation of an instance of the role like it were an instance of the inner class, for example:

```
Printer laser = new Printer();
AccountedPerson chris = new AccountedPerson();
CertifiedPerson sergio = new CertifiedPerson();
laser.new User(chris);
laser.new SuperUser(sergio);
```

The first instructions create a `Printer` object `laser` and two objects, `chris` (an `AccountedPerson`) and `sergio` (a `CertifiedPerson`). `chris` becomes a normal `User` while `sergio` becomes a `SuperUser`. Indeed, the last two instructions define the roles of these two objects with respect to the created `Printer`. Note that all the constructors of role implementations have a first (implicit) parameter which must be bound to the player of the role and whose value becomes the value of `that`.

Playing a role

When an object is seen under the perspective of a role, it has a specific state for it, which is different from the player's one. This state is associated with the pair of objects "institution" and "player". It is the state of the interaction between the caller and the callee object and it evolves as a consequence of the invocation of methods on the role (or on other roles of the same institution as we have seen in the running example). In the printer example the variable `counter` of `User` keeps track of the number of printed pages for each different user. We will come back to this a little ahead.

When an object uses the methods offered by a role, it should be able to invoke them without 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 should only specify in which role it is invoking the method. For example, if an `AccountedPerson` is a `User` and it has to print something, it must be able to invoke the method `print` on the `AccountedPerson` as a `User` without referring to the role instance. Note that this does not exclude the possibility of assigning the reference to a role instance to a variable then using the variable for invoking the role methods (see the variable user in the code below). Roles belong always to an institution. Hence, an object can play at the same moment the same role more than once, albeit in different institutions. Instead, we do not consider the case of an object playing the same role more than once in the same institution. An object can play several roles in the same institution. In order to specify the role under which an object is referred, we evocatively use the same terminology used for casting by Java: we say that there is a casting from the object to the role. However, to refer to an object in a certain role, both the object and the institution where it plays the role must be specified, thus reflecting the foundation property. We call this methodology *role casting*. Role casting is a means for stating that an object will act according to the powers that allow it to interact in a given institution. In the following the two Users invoke method print on laser. Notice that the page counter is maintained in the role state and persists through different calls to methods performed by a same player towards the same institution as long as it plays the role.

```
((laser.User) chris).print(job1);
((laser.SuperUser) sergio).print(job2);
System.out.println("Chris has printed " +
  ((laser.User) chris).getPrintedPages() + " pages");
System.out.println("The printer laser has printed a total of " +
  ((laser.SuperUser) sergio).getTotalPrintedPages() + " pages");

User user = ((laser.User) chris);
user.print(job3);
System.out.println("Chris has printed " +
  ((laser.User) chris).getPrintedPages() + " pages");
```

Supposing that job1 consists of ten pages, job2 of twenty pages and job3 of fifteen, the first output operation will print ten, the second one thirty (the sum of the lengths of job1 and job2), the third one twentyfive (the sum of job1 and job3).

By maintaining a state, a role can be seen as realizing a *session-aware interaction*, in a way that is analogous to what done by cookies or Java sessions for JSP and Servlet. So in our example, it is possible to visualize the number of currently printed pages by the user chris.

Since an object can play multiple roles, the same method will have a different behavior, depending on the role which the object is playing when the method is invoked. However, there will be no conflict among roles, since only the powers of one role at a time can be exercised. To play a role it is sufficient to specify which is the role of a given object we are referring to. In the next example sergio becomes also a normal User of laser, besides being a SuperUser, since a CertifiedPerson is also an implementation of the interface Accounted:

```
laser.new User(sergio);
((laser.SuperUser) sergio).print(job4);
((laser.User) sergio).print(job5);
```

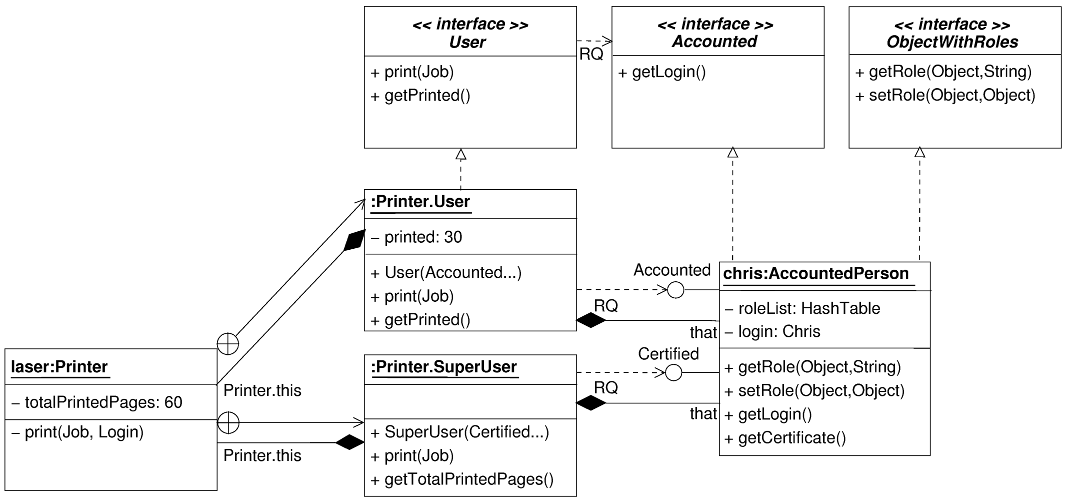

Figure 3: The UML diagram representing our running example.

Notice that in this case two different sessions will be kept: one for `sergio` as normal `User` and the other for `sergio` as `SuperUser`. Only when it prints its jobs as a normal `User` the page counter in the role instance is incremented.

A role instance can be left by a player or transferred to another player satisfying the requirements. In the first case, the invariant imposing the foundation of a role on its player is violated. The invocation of a method on such a role instance (which is possible since the role instance could have been assigned to a variable before the player gives up its role or it is destroyed) gives raise to an exception. However, we do not deal with these issues in this work, see [34] for a discussion.

4 ROLE REPRESENTATION AND TRANSLATION INTO PURE JAVA

In this section we will present the translation of powerJava in pure Java illustrating it with a UML diagram, showing a portion of our example, see Figure 3.

For what concerns the translation phase into pure Java, this is done by means of a pre-compilation phase. The pre-compiler has been implemented by means of the tool javaCC, provided by Sun Microsystems.

First of all, role requirements are an interface, specifying which methods must be defined in a class whose instances play the role. Powers, instead, are a new concept related to the `role` construct, representing the methods acquired by role players: the abstract signatures of power specifications are translated into an interface too, related to the interface requirements by the relation RQ in Figure 3:

```
interface User {
  int print(Job job);
  int getPrintedPages();
}

interface Accounted {
  Login getLogin();
}

interface SuperUser {
  int print(Job job);
  int getTotalPrintedPages();
  void resetPrinterCounter(User user);
 }

interface Certified extends Accounted {
  Certificate getCertificate();
}
```

Second, the implementation of roles inside the institution is translated as inner classes, which implement the interface which results from the translation of the role power specification. Inner classes express the fact that the namespace of the institution is visible from the role implementation. For instance, in Figure 3, in the institution Printer the roles User and SuperUser are inner classes. The fact that inner classes belong to the namespace of the outer class is represented in UML by the arrow with a plus sign within a circle at the end attached to the namespace.

The difference with inner classes is that while an inner class can be instantiated given an instance of its outer class, an inner class defining the implementation of a role does not create an object which exists independently also from the object which plays the role. In other words, the instance of the role must be connected both with its player and its institution. In Figure 3 the references to such unnamed objects corresponding to the role instances are respectively represented by the composition arrows with the labels *Printer.this* and *that*. The following is an excerpt of the translation of the class Printer:

```
class Printer {
  final static int MAX_PAGES_PER_USER;
  private int totalPrintedPages = 0;
  private void print(Job job, Login login) {
    totalPrintedPages += job.getNumberPages(); // performs printing
  }
  private boolean validCertificate(Certificate cert) {
    // checks the certificate cert
  }
  class UserPower implements User  {
    Accounted that;
    public UserPower(Accounted that) {
      this.that = that;
      ((ObjectWithRoles)this.that).setRole(Printer.this, this);
    }
    // role's fields and methods ...
  }

  class SuperUserPower implements SuperUser {
    Certified that;
    public SuperUser(Certified that) {
      this.that = that;
      ((ObjectWithRoles)this.that).setRole(Printer.this, this);
      //first, verify the identity of the player
      if (!validCertificate(this.that.getCertificate()))
        throw new Exception("You are not allowed to enter this role");
    }
    // role's fields and methods ...
  }
}
```

When an inner class implements a role, the role name specified by the `definerole` keyword is simply added to the interfaces implemented by the inner class. The correspondence between the player and the role instance, represented by the construct `that`, is pre-compiled in a field called `that` of the inner class. This field is automatically initialized by means of the constructors of role classes which are extended by the pre-compiler by adding a first parameter to pass the suitable value. The constructor also adds to the role player referred by `that` a reference to the role instance. The remaining link between the instance of the inner class and the outer class defining it is provided automatically by the language Java (e.g., `Printer.this`). Note that this translation also explains why the keyword `playedby` can be followed by just one identifier. The reason is that it would not be possible to assign a correct static type to `that`. One possibility for overcoming this limitation would be to rely on union types, as proposed, for example, by Igarashi and Nagira [24].

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. In this way, in role casts the role instance can be retrieved from its player given the role name and a reference to the institution. This is obtained by adding, at pre-compilation time, to every class a structure for book-keeping its role instances. This structure can be accessed by the methods whose signature is specified by the `ObjectWithRole` interface (see Figure 3). Since every object can play a role, it is worth noticing that the ideal solution would be that the `Object` class itself implements `ObjectWithRole`:

```
interface ObjectWithRoles {
  public void setRole(Object inst, Object role);
  public Object getRole(Object inst, String roleName);
}
```

The two methods that are introduced by the pre-compiler are `setRole` and `getRole` which, respectively, adds a role instance to an object, specifying where the role is played, and returns the role instance played in the institution passed as parameter together with the role name. Further methods can be added for leaving a role, transferring it, *etc.*:

```
class AccountedPerson implements Accounted, ObjectWithRoles {
  private java.util.Hashtable roleList = new java.util.Hashtable();
  public void setRole(Object inst, Object role) {
    roleList.put(inst.hashCode() + role.getClass().getName(), role);
  }
  public Object getRole(Object inst, String roleName) {
    return roleList.get(inst.hashCode() +
      inst.getClass().getName() + "$" + roleName);
  }
  // class' fields and methods ...
}
```

The `setRole` and `getRole` methods make use of a private hash-table `roleList`. As key in the hash-table we use the institution instance address and the name of the inner class. As an example, the class `AccountedPerson` plays the role `User` via the interface `Accounted`. So its instances will have a hash-table that keeps the many roles played by them. Role casting is pre-compiled using these methods. The expression referring to an object in its role (an `AccountedPerson` as a `User`, e.g., `(laser.User) chris`) 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 `chris.getRole(laser, "UserPower")`. The string `"UserPower"` is provided because in our solution the name of the role class is used as a part of the key of the hash-table:

```
((Printer.UserPower)chris.getRole(laser, "UserPower")).print(job1);
((Printer.SuperUserPower)sergio.getRole(laser, "SuperUserPower")).
    print(job2);
```

With respect to Java, additional checking is introduced to verify the consistency of the newly introduced constructs. For example, allowing classes prefixed by variables (e.g., `laser.User`), in the style of the Scala language [35], in the role cast constructs, introduces ambiguities which pure Java is not aware of.

5 RELATED WORK

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 [40], 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, equality fundamental notion: that of roles. Although definitions of the concept of role 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 of role: (a) names for association ends, like in UML or in Entity-Relationship diagrams; (b) dynamic specialization, like in the Fibonacci [2] programming language; (c) adjunct instances, like in the DOOR programming language [44] or ObjectTeams [23]. The two last views are more relevant for modelling roles in programming languages.

We stick to the adjunct instance perspective, with an important difference with most previous work, with the partial exception of [23, 30, 42]: the role instance is always associated with both the player of the role and the callee object which the role belongs to.

Most other works on roles in programming languages adopt a different perspective: roles are seen as a way to extend the behavior of an object and not as a way to model how an object offers different possibilities of interaction to different kind of players. Thus, there is a deep difference with our approach: we pass from a *player-centered* vision of roles to an *organizational-centered* one. The different perspective is also signalled by the terminology used. When other works use the phrase "the role of an object" they mean "the role played by a (caller) object" (since there is no explicit context offering that role). Instead, by "the role of an object" we mean the role a callee offers to play to a caller object in order to enable the caller to interact with the callee. In the *player-centered* approach, the printer example we propose could be modelled only by adding roles like user and superuser to instances of the class person without any systematic relation with the class printer, thus preventing the possibility that the role and the institution share their namespace. The reason of this difference is that these proposals, even when they share some similarities with our work, aim at solving different kind of practical problems than allowing callee objects to exhibit specific behaviors to specific callers.

In this respect our perspective is more similar to the use of role in security, e.g., in the role based access control (RBAC) model [38]. In RBAC roles are used to distinguish different set of authorizations to interact with the resources of a system, and sessions are mappings between a user and an activated subset of roles that are assigned to the user. However, in our model, methods of roles do not exist without the role offering them. For this reason attributes and operations are described by means of classes. In the RBAC model, instead, roles are only groupings of rights concerning operations defined directly in the system, and the operations have the same meaning for all roles. Since role classes provide a different instance for each caller which plays the role, they can represent the state of the interaction between the caller object playing a role and the callee which offers the role. Hence, there can be more than one session for each caller object playing different roles. This is again different from the RBAC model where sessions are associated with users and not with role instances.

Many works on the introduction of roles in programming languages [2, 16, 21, 37] consider roles as a specialization of classes, e.g., a customer is seen as a (dynamic) specialization of the class person. In our example, user and superuser would have been subclasses of person. This methodology, as Steimann [40] notices, does not capture the intuition that a role like customer can be played both by a person and by an organization. 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.

Multiple inheritance does not help, since it becomes impossible to have context dependent access, as Dahchour *et al.* [16] notice. Fibonacci [2], e.g., is a language which introduces a hierarchy of role types to specialize an object class. This language also supports a radical view of separation of concerns by imposing that the interaction with an object always passes through a role.

Roles as specializations prevent realizing that roles are always associated not only with a player, but to an institution, which defines them, too. This intuition sometimes implicitly emerges also in these frameworks: in [37] the authors say "a role is *visible* only within the scope of the specific application that created it", but such contexts are not first class citizens like institutions are in our model.

Kristensen and Osterbye [26] recognize the fact that a role depends on its player but they fail to recognize the dependency of a role from the institution. Moreover, they consider roles as a form of specialization, albeit one distinguishing the role as an instance related to, but separated from, its player. As a consequence, the properties of the role include the properties inherited from its player. This idea conflicts with our position we adopt from Steimann [40]: roles are partial descriptions of behavior, thus they shadow the other properties of their players.

Wong *et al.* [44] recognize that roles are adjunct instances. They introduce a parallel role class hierarchy connected by a player relationship to the object class hierarchy. Moreover, in their model a role player qualification specifies which classes can play the role. This corresponds to our idea of associating requirements to a role, which we model as interfaces. However, like many of the previous approaches, Wong *et al.* [44] fail to capture the intuition that a role depends on the context defining it. Another major difference with their approach is that we reject the method lookup as delegation. This methodology has a troublesome implication: when a method is invoked on some object in one of its roles, the meaning of the method can change depending on all the other roles played by the object. We do not consider this as a desirable feature in the context of a language like Java.

Our approach shares the idea of gathering roles inside wider entities with Object Teams [23] and Caesar [30]. These languages emerge as refinements of aspect oriented languages aiming at resolving practical limitations of aspect programming. In contrast, our language starts from different practical problems and introduce an ontologically founded definition of roles. Differently from [23, 30] in our work the method calls make explicit in which role the caller is invoking the method. Moreover, it is not only the meaning of methods which changes, but also the possible methods that can be invoked. If one would like to integrate the aspect paradigm within our view of object oriented programming, the natural place would be use aspects to model the environment where the interaction between objects happens. Consider the `within` construct in Object Teams/Java [23] which specifies aspects as the context in which a block of statements has to be executed. Since, when defining the interaction possibilities of an object it is not possible to foresee all possible contexts in which the interaction happens, the effect on the environment can be better modelled as a crosscutting concern. Thus, aspect programming is a complementary approach with respect to ours.

Other works which recognize the dependence of roles from a context are [42] and [27]. In Tamai [42], the concepts of *context* (roughly corresponding to our institution) and *role* are related in the language *EpsilonJ*. However, the approach that we propose better adheres to the Java programming style because it relies onto features that are already in this language (roles are implemented using inner classes and institutions are classes), basically adding just the concept of role interface and role casting. In contrast, in *EpsilonJ* roles and contexts are introduced as new constructs, and their relation with classes and objects is not explicit. Moreover, their proposal is not implemented in Java but in Ruby, limiting the applicability of their approach and not taking into account typing issues.

Lee and Bae [27] introduce the notion of role system and model it by means of a special class with the function of maintaining the coherence among roles and among roles and their players. However, they propose a pattern and not a language.

Baumer *et al.* [11] propose the role object pattern to solve the problem of providing context-specific views of the key abstractions of a system, since different context-specific views cannot be integrated in the same class. They propose to model context-specific views as role objects which are dynamically attached to a core object, thus forming what they call a subject. This adjunct instance should share the same interface as the core object, so the role cannot have methods different from those of their players.

Mossè [32] presents several patterns related to roles, but none of them considers the problem of roles belonging to institutions for offering access to them.

Molina *et al.* [31] revise the OOram methodology for modelling roles, translating it to UML. Even if also in this case roles do not belong to institutions, there are some similarities in that role diagrams specify the interaction among a set of roles.

Even if the notion of role is not mentioned explicitly, in Aksit *et al.* [1] the notion of abstract communication types has some similarities with our approach, in that communication can be specifies separately from the core behavior of an object. However, they use a reflection based mechanism which makes the approach very flexible but difficult to control. Methods are associated with a set of conditionals acting as guards which prevent inappropriate methods to be called in certain states of the object. Our approach instead sticks to the philosophy of Java trying to overcome the underlying vision of object orientation.

Our notion of role, as a double-face interface, bears some similarities with Traits [39] and Mixins. However, Mixins are distinguished because they are used to extend classes and not instances, with a few exceptions like, e.g., [12]. In contrast, roles extend instances.

Similarities and differences can be found also in our ontological definition of roles with respect to other approaches. Our main aim is to stick to a widely acceptable definition in knowledge representation, so that we left out controversial issues like roles playing roles, which we discuss in [15]. For example, Masolo *et al.* [29] accept the notion of foundation, of definitional dependence, albeit a weaker one, where the definition of a role must use the concept of the institution. Loebe [28] also considers roles as dependent on contexts, but he does not stress the fact that roles can be seen as different ways of interacting with an object; Viganò and Colombetti [43] also consider powers as an essential feature of roles. Moreover, we are inspired from Guarino and Welty [22] in considering roles as antirigid and dynamic. Also Steimann [40] is an important source of inspiration, when he highlights that a role can be played by different kinds of actors. However, differently from Steimann in our model roles cannot be reduced only to interfaces and have an identity, in the sense that they can become the value of a variable, even if role instances are not independent as they are founded.

The six problems we discussed in the introduction have been already identified, even if separately, but they have been addressed only in partial or indirect ways.

Programming languages like Fickle [18] address the second and third problem by means of dynamic reclassification: an object can change class dynamically, and its operations change accordingly. However, Fickle does not represent the dependence of attributes and operations from the interaction, and all the subclasses share the same interface. StateJ [17] offers different implementations of methods according to a feature called the state of the object. However, an object has only one state and the state does not depend on the caller of a method and different callers or caller types cannot correspond to different states. Aspect programming focuses too on enhancing the modularity by means of crosscutting concerns, but it is less clear how it addresses all the six concerns.

Some patterns partially address the above mentioned issues; for example, the strategy design pattern [19] allows objects to dynamically change the implementation of a method. However, it is complex to implement and it does not address the problem of having different methods offered to different types of callers and of maintaining the state of the interaction between caller and callee. So more patterns must be combined with additional problems such as the lack of modularity and increased complexity.

Sessions are dealt with by distributed programming constructs, like the JSP, Servlets or Enterprise Java Beans, but these solutions do not consider the other problems of having different methods and states offered to different types of callers, beside the fact that sessions are not integrated in the method invocation syntax. Rather, interaction happens often only via a web service.

Sessions are also considered in the agent oriented paradigm, which bases communication on protocols [25]. In agent orientation a protocol is the specification of the possible sequences of messages exchanged between two agents, and not simply an interface. Since not all sequences of messages are legal, the state of the interaction between two agents must be maintained in a session. Moreover, not every agents can interact with other ones using whatever protocol. Rather, the interaction is allowed only between agents playing certain roles. Thus, an agent displays different possibilities of interaction to different types of interactants by playing different roles in the interaction with them, and it can even play different roles at the same time in the interaction with the same agent. However, the notion of role in multi-agent systems is rarely related with the notion of session of interaction ([36]).

6 SUMMARY

In this paper we identify some problems in the current view of object orientation, namely, the facts that callers of methods are not made explicit, attributes and operations associated with callee objects do not depend on the caller's identity or class, and that there is no session keeping track of the interaction between a callee and a caller object. To overcome these limitations, we introduce the view on roles adopted at higher level in access control in web services and we transfer it at lower level in object oriented programming languages. We base on an ontological analysis of the notion of role to understand which are its properties.

We introduce this model of roles in an extension of Java, called powerJava. First, roles are implemented by classes which can be instantiated only in presence of an instance of the player of the role (caller) and of an instance (callee) of the class representing the institution (foundation). The role instance represents the session of the interaction between caller and callee. Second, the implementation of a role is included in the class definition of the institution the role belongs to using inner classes (definitional dependence). Thirdly, the players of roles have powers since methods of roles can access private fields and methods of the institution they belong to and of the other roles of the same institution (institutional empowerment). Finally, to express the fact that an object can be seen in one of the roles it plays we introduce the notion of role casting.

As discussed in Section 4, powerJava is more than just the transfer of a pattern in a language. Additional benefits come from the introduction of roles directly in the language, like the management of role instance invariants, the introduction of suitable exceptions, or the fact that roles can be type checked, partly by delegating this task to the Java translation.

Details about the implementation can be found in [4] and the pre-compiler is available at `http://www.powerjava.org`.

Roles allow the programmer to adopt different interaction possibilities between callers and a callee, which do not exclude the traditional direct interaction with the object when roles are not necessary. Other possibilities like sessions shared by different objects are not considered for space reasons.

First of all, an object can interact with another one by means of the role offered by it. This is, for instance, the case of `chris` being a `User` of `laser` in Section 3.

Second, a caller object (e.g., `sergio` in our example) can interact in two different roles with a callee object. This situation is used when a callee object implements two different interfaces for interacting with it, which have methods (like `print`) with the same signature but with different meaning. In our model the methods of the interfaces are implemented in the roles offered by the objects to interact with them. The role represents also the different sessions of the interaction with the same object.

Third, two caller objects can interact with each other by means of the (possibly different) roles of an institution. This is the original case powerJava has been developed for [9]: achieving separation of concerns, taking apart the core behavior of a class from the dynamically acquired behavior in an unforeseen context; in that paper, we used as a running example the well-known five philosophers scenario. The institution is the table, at which philosophers are sitting and coordinate to take the chopsticks and eat since they can access the state of each other. s

Fourth, two objects can interact with each other, each playing a role offered by the other. This is often the case of interaction protocols: e.g., an object can play the role of *initiator* in the Contract Net Protocol if and only if the other object plays the role of *participant*.

Our view of roles inspires a new vision of the object oriented paradigm, whose object metaphor has been accepted too acritically and it has not been subject to a deep analysis. In particular, it adopts a naive view of the notion of object and it does not consider the analysis of the way humans conceptualize objects performed in philosophy and above all in cognitive science [20]. In particular, cognitive science has highlighted that properties of objects are not objective properties of the world, but they depend on the properties of the agent conceptualizing the object: objects are conceptualized on the basis of what they “afford” to the actions of the entities interacting with them. Thus, different entities conceptualize the same object in different ways. We translate this intuition in the fact that an object offers different methods according to which type of object it is calling it: the methods offered (the powers of a role) depend on the requirements offered by the caller. This perspective is analysed in [6, 7].

powerJava has also been used as a coordination language dealing with concurrency [9]. We discuss how to use powerJava to implement protocols in multi-agent systems in [5].

Future work concerns, first of all, the relational nature of roles which often come into pairs, like client/customer in a business exchange, manager/bidder in a negotiation protocol, *etc.* In [10] we use roles to model how objects can participate to relationships and, thus, acquire new properties and behaviors. We add roles to the existing relationship as attribute and relationship object patterns proposed by Noble and Grundy [33]. Second, some issues concerning the translation of powerJava in Java must be deepened. In particular, type checking should be clarified, and introduced also during the pre-compilation phase instead of relying only on the Java compiler, so that more explicit errors can be signalled to the user. Finally, the lifecycle of roles should be studied (see, e.g., Odell *et al.* [34]), and a corresponding systems of exceptions developed to signal which are the possible errors related to roles at runtime.

ACKNOWLEDGEMENTS

Matteo Baldoni has partially been funded by the 6th Framework Programme project REWERSE number 506779, and by MIUR PRIN 2005 “Specification and verification of agent interaction protocols” national project. Guido Boella has partially been founded by the MIUR PRIN 2006 project “Documentalità. Ontologie e tecnologie per la cittadinanza e la democrazia”.

The authors thank the anonymous reviewers for their useful suggestions.

REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Procs. of ECOOP ’93 Workshop on Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 152–184, Berlin, 1994. Springer Verlag.

[2] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Procs. of Very Large DataBases (VLDB’93)*, pages 39–51, 1993.

[3] F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 33–70. Springer, Berlin, 2003.

[4] M. Baldoni, G. Boella, and L. van der Torre. Social roles, from agents back to objects. In *Procs. of From Objects to Agents Workshop (WOA’05)*, Bologna, 2005. Pitagora.

[5] M. Baldoni, G. Boella, and L. van der Torre. Bridging agent theory and object orientation: Interaction among objects. In *Procs. of PROMAS'06 workshop at AAMAS'06*, 2006.

[6] M. Baldoni, G. Boella, and L. van der Torre. Interaction among objects via roles: sessions and affordances in powerJava. In *Procs. of Principles and Practice of Programming in Java (PPPJ'06)*, pages 188–193, New York (NY), 2006. ACM.

[7] M. Baldoni, G. Boella, and L. van der Torre. Modelling the interaction between objects: Roles as affordances. In *Procs. of Knowledge Science, Engineering and Management, KSEM'06*, volume 4092 of *LNCS*, pages 42–54. Springer, 2006.

[8] M. Baldoni, G. Boella, and L. van der Torre. powerJava: ontologically founded roles in object oriented programming language. In *Procs. of ACM Symposium on Applied Computing (SAC'06), Track Object Oriented Programming Languages and Systems (OOPS'06)*, pages 1414–1418. ACM, 2006.

[9] M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct: Introducing powerJava. *Electronic Notes in Theoretical Computer Science*, 150(1):9–29, 2006.

[10] M. Baldoni, G. Boella, and L. van der Torre. Relationships meet their roles in object oriented programming. In *Procs. of FSEN'07*, 2007.

[11] D. Baumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern. In *Procs. of PLOP'02*, 2002.

[12] L. Bettini, V. Bono, and S. Likavec. A core calculus of mixin-based incomplete objects. In *Procs. of Foundations and Developments of Object Oriented Languages Workshop (FOOL'04)*, pages 29–41, 2004.

[13] G. Boella and L. van der Torre. An agent oriented ontology of social reality. In *Procs. of Formal Ontologies in Information Systems (FOIS'04)*, pages 199–209, Amsterdam, 2004. IOS Press.

[14] G. Boella and L. van der Torre. A foundational ontology of organizations and roles. In *Procs. of DALT'06 workshop at AAMAS'06*, 2006.

[15] G. Boella and L. van der Torre. The ontological properties of social roles in multi-agent systems: Definitional dependence, powers and roles playing roles. *Artificial Intelligence and Law*, 2007.

[16] M. Dahchour, A. Pirotte, and E. Zimanyi. A generic role model for dynamic objects. In *Procs. of Conference on Advanced Information Systems Engineering (CAiSE'02)*, volume 2348 of *LNCS*, pages 643–658. Springer, 2002.

[17] F. Damiani, E. Giachino, P. Giannini, and E. Cazzola. On state classes and their dynamic semantics. In *Procs. of International Conference on Software and Data Technologies (ICSOFT'06)*, pages 5–12. INSTICC, 2006.

[18] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{II}$. *ACM Transactions On Programming Languages and Systems*, 24(2):153–191, 2002.

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

[20] J. Gibson. *The Ecological Approach to Visual Perception.* Lawrence Erlabum Associates, New Jersey, 1979.

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

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

[23] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Procs. of Net.ObjectDays*, 2002.

[24] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *Procs. of ACM symposium on Applied Computing (SAC '06)*, pages 1435–1441, New York, NY, USA, 2006. ACM.

[25] T. Juan, A.R. Pearce, and L. Sterling. ROADMAP: extending the GAIA methodology for complex open system. In *Procs. of Autonomous Agents and Multiagent Systems Conference (AAMAS'04)*, pages 3–10, 2002.

[26] B.B Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[27] Joon-Sang Lee and Doo-Hwan Bae. An enhanced role model for alleviating the role-binding anomaly. *Software: Practice and Experience*, 32(14):1317 – 1344, 2002.

[28] F. Loebe. Abstract vs. social roles - a refined top-level ontological analysis. In *Procs. of AAAI Fall Symposium Roles'05*, pages 93–100. AAAI Press, 2005.

[29] C. Masolo, L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi, and N. Guarino. Social roles and their descriptions. In *Procs. of Conference on the Principles of Knowledge Representation and Reasoning (KR'04)*, pages 267–277. AAAI Press, 2004.

[30] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Procs. of International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100. ACM Press, 2004.

[31] J.G. Molina, M.J. Ortin, B. Moros, and J. Nicolas. Transforming the OOram three-model architecture into a UML-based process. *Journal of Object Technology*, 1(4):119–136, 2002.

[32] F.G. Mosse. Modeling roles: A practical series of analysis patterns. *Journal of Object Technology*, 1(4):27–37, 2002.

[33] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Procs. of Technology of Object-Oriented Languages and Systems (TOOLS)*, 1995.

[34] J. Odell, H. Van Dyke Parunak, S. Brueckner, and J. Sauter. Changing roles: Dynamic role assignment. *Journal of Object Technology*, 2(5):77–86, 2003.

[35] M. Odersky. The Scala Experiment – can we provide better language support for component systems? In *Procs. of ACM Symposium on Principles of Programming Languages*, pages 166–167, 2006.

[36] A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, 2005.

[37] M.P. Papazoglou and B.J. Kramer. A database model for object dynamics. *The Very Large DataBases Journal*, 6(2):73–96, 1997.

[38] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 2:38–47, 1996.

[39] N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Procs. of European Conference on Object Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274, Berlin, 2003. Springer.

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

[41] F. Steimann and P. Mayer. Patterns of interface-based programming. *Journal of Object Technology*, 4(5):75–94, 2005.

[42] T. Tamai. Evolvable programming based on collaboration-field and role model. In *Procs. of International Workshop on Principles of Software Evolution (IW-PSE'02)*, pages 1–5. ACM, 2002.

[43] F. Viganò and M. Colombetti. Specification and verification of institutions through status functions. In *COIN@AAMAS'06 Workshop*, 2006.

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

ABOUT THE AUTHORS

Matteo Baldoni received his Ph.D. in Computer Science in May 1998 from the University of Torino. He is currently associate professor at the Department of Computer Science of the University of Torino, Italy. He has a background in computational logic, modal and nonmonotonic extensions of logic programming, multimodal logics, reasoning by actions and change. His current research interests include issues in communication protocol design and implementation, conformance and interoperabilty for agents and web services, agent programming languages, personalization by reasoning in the semantic web. He organized the last three editions of the Declarative Agent Languages and Technologies International Workshop. He can be reached at baldoni [at] di.unito.it. See also http://www.di.unito.it/~baldoni.

Guido Boella received the PhD degree at the University of Torino in 2000. He is currently professor at the Department of Computer Science of the University of Torino. His research interests include multi-agent systems, in particular, normative systems, institutions and roles using qualitative decision theory. He organized the first two workshops on normative multi-agent systems, on coordination and organization, and the first AAAI Fall Symposium on roles. guido [at] di.unito.it. See also http://www.di.unito.it/ guido.

Leendert W. N. van der Torre received the PhD degree from Erasmus University Rotterdam in 1997. He is currently full professor at the University of Luxembourg. He has developed the so-called input/output logics and the BOID agent architecture. He has written over one hundred scientific papers, and he has organized the first two workshops on normative multi-agent systems and the firsts two workshop on coordination and organization. His current research interests include deontic logic, qualitative game theory and coordination in normative multi-agent systems. leon.vandertorre [at] uni.lu. See also http://agamemnon.uni.lu/ILIAS/vandertorre.