powerJava: Ontologically Founded Roles in Object Oriented Programming Languages

Matteo Baldoni
Dipartimento di Informatica
Università degli Studi di Torino
baldoni@di.unito.it

Guido Boella
Dipartimento di Informatica
Università degli Studi di Torino
guido@di.unito.it

Leendert van der Torre
CWI Amsterdam and Delft university of Technology
torre@cwi.nl

ABSTRACT

In this paper we introduce a new view on roles in Object Oriented programming languages, based on an ontological analysis of roles. A role is always associated with an object instance playing the role and also to an object instance (its institution) which represents its context. The definition of a role depends on the definition of the institution. This property allows to endow role-players with powers that can modify the state of the institution and of the other roles defined in it. As an example, we introduce a role construct in Java, where the abolve features are interpreted as follows. Roles are implemented as classes, which can be instantiated only in presence of an instance of the player and of an instance of an institution. The definition of a class implementing a role is included in the class of the institution, the role belongs to. Powers are methods which can access private fields and methods of the institution they belong to, and of the other roles of the same institution.

1. INTRODUCTION

The concept of role is used quite ubiquitously in Computer Science: databases to multiagent systems, from conceptual modelling to programming languages. According to Steimann [0], 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 role concept abound in the literature, Steimann maintans 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 [0] programming language; (c) Adjunct instances, like in the DOOR programming language [0]. The two last views are more relevant for modelling roles in programming languages. Both of them have pros and cons. For example, dynamic specialization captures the dynamic relation between a class and a role which can be played by it (e.g., a person can become a student), but it less easily models the intuition that roles can have their own state (e.g., an employee has a different phone number than the person playing that role). In contrast, roles as adjunct instances can obviously have their own state, but they may pose problems when role instances are detached from the object which plays the role.

There is a wide literature on the introduction of the notion of role in programming languages. However, most works, starting from Bachman and Daya [0]'s revision of database models, extend programming languages with roles starting from practical considerations. In contrast, the research question of this paper is the following: How to introduce in an OO programming language a notion of role which is ontologically well founded? We refer to the ontological analysis of the notion of role made in [0, 0, 0]. According to that proposal, roles have the following properties:

- Roles are always associated both with an object instance playing the role, and to another object instance which constitutes the context of the role and which we call the *institution*.
- The definition of a role depends on the definition of the institution which constitutes its context.
- This second property allows to endow players of roles with powers to modify the state of the institution and of the other roles of the same institution.

For example, the role *student* has a *person* as its player and it is always a student of a *school*, a *president* is always the president of an *organization*, a *customer* can be played by a *person* or an *organization*, and it is always a *customer* of an *enterprise*. In contrast, almost all current approaches focus only on the relation between the role and its player. The methodology we follow is to introduce a new programming construct in a real programming language, Java, one of the most used OO languages and one of the most principled. To prove its feasibility, we translate the new language, called *powerJava*, to pure Java by means of a precompilation phase.

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

In Section we summarize the ontological definition of roles while, in Section , we introduce roles Java with powerJava. Related work and conclusion end the paper.

SAC'06 April 23-27, 2006, Dijon, France

2. FOUNDATION, DEFINITIONAL DEPENDENCE, AND POWERS

The distinguishing features of roles in [0, 0, 0] are their foundation, their definitional dependence from the institution they belong to, and the powers attributed to the role by the institution. Consider the roles student and teacher. A student and a teacher are always a student and a teacher of some school. Without the school the roles do not exist anymore: e.g., if the school goes bankrupt, the actors (e.g. a person) of the roles cannot be called teachers and students anymore. The institution (the school) also specifies the properties of the student, which extend the properties of the person playing the role of student: the school specifies its enrollment number, its email address, its scores at past examinations, and also how the student can behave. For example, the student can give an exam by submitting some written examination. A student can make the teacher evaluate its examination and register the mark because the school defines both the student role and the teacher's role: the school specifies how an examination is evaluated by a teacher, and maintains the official records of the examinations. Otherwise the student could not have an effect on the teacher. But in defining such actions the school *empowers* the person who is playing the role of student: without being a student the person has no possibility to give an examination and make the teacher evaluate it.

This example highlights the following properties that roles have in our model [0, 0, 0]:

- *Foundation*: a (instance of) role must always be associated with an instance of the institution it belongs to (see Guarino and Welty [0]), besides being associated with an instance of its player.
- *Definitional dependence*: The definition of the role must be given inside the definition of the institution it belongs to. This is a stronger version of the definitional dependence notion proposed by Masolo *et al.* [0], where the definition of a role must use the concept of the institution.
- *Institutional empowerment*: the actions defined for the role in the definition of the institution have access to the state and actions of the institution and of the other roles: they are powers.

Moreover, as noticed in [0], 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, [0] highlights that a role can be played by different kinds of actors. 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 actors. This requirement is in line with UML, which relates roles and interfaces as partial descriptions of behavior.

This last property compels to avoid modelling roles as dynamic specializations as, e.g., [0, 0] do. If customer were a subclass of person, it could not be at the same time a subclass of organization, since person and organization are disjoint classes. Symmetrically, person and organization cannot be subclass of customer, since a person can be a person without ever becoming a customer.

3. INTRODUCING ROLES IN JAVA: POWERJAVA

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

Analogously to classes and interfaces in OO, we distinguish the role implementation in an institution from the specification of both its powers and requirements. A role implementation should implement the power specifications, while a player should implement the requirement specifications. In other words, first a role is defined by means of an abstract definition, similar to a Java interface (keyword `role`), that specifies what is requested to play it and what is offered by it (keyword `playedby`).

The implementation of a role is done by means of a Java inner class, which implements the powers (keyword `realizes`). Since Java inner classes allow a class to belong to the namespace of another class, we use them to give powers to roles in institutions. Moreover, implementing a role definition as an inner class of an outer class defining an institution parallels exactly the definitional dependence. The association of a role instance with an institution instance can be dealt with the implicit reference (in Java) to an inner class from its outer class. So we are left only to deal explicitly with the association of a role instance with a player instance, to complete foundation. Finally, seeing an object under a role is paralleled with type casting in Java.

From a higher perspective, we model roles as instances of role classes, which can be associated at runtime with objects which can play a role. However, roles are a special kind of objects, and instances of role classes do not exist on their own, but they are always associated with an object instance of its player and with an object instance of the related institution. The relations of a role with these two instances are different. Concerning the former relation, the player of the role is an object whose properties and behavior are extended when it is seen under the perspective of the role. Moreover, the role does not affect the core behavior. We keep distinct the player object itself from the role that it is playing: for using a role power it is necessary to specify explicitly the role, otherwise the player object own methods are invoked (object methods are not implicitly overridden). Concerning the object instance, which represents the institution, and the roles supplied by it, a role is enabled to access the institution's own state and the state of the other roles via its methods; thus, the role's behavior can effect the institution's behavior. Accessing the institution's state is possible only if the classes defining it and its roles are connected. This is what it is called definitional dependence and it requires that the role class belongs to the namespace of the institution class.

Finally, the constraint of foundation requires that the creation of a role instance involves both an institution instance and an object instance. An object can play several roles and also the same role in different institutions at the same time. Hence, the role under which a player is seen must be specified using also the institution instance. Of course, when a role instance is explicitly referred to, this is not necessary.

3.1 Specification of powers and requirements

Specifying a role implies specifying both what is required

```
interface StudentReq //Student's requirements
{ String getName();
  int getSocialSecNumber(); }

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

interface TeacherReq  // Teacher's requirements
{ String getName();
  int getSocialSecNumber();
  int getQualificationNumber();
  int read(HomeWork hwk); }

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

Figure 1: Specification of powers and requirements.

to play it and which powers the player acquires in the institution in which the role is implemented. In order to make role systems reusable, it is necessary that a role is not played by a class only. For Steimann and Mayer [0], roles define a certain behavior (or protocol) demanded in a context independently of how or by whom this behavior is to be delivered (and, we add, roles also empowers the player in the context). Thus, roles must be specified independently of the particular classes playing the role, so that the objects which can play it might be of different classes and can be developed independently of the implementation of the role. This is a form of polymorphism. In fact, role requirements are specified as an interface, that can be implemented by different classes. Thus, a role specification has to list the methods offered to objects playing the role (*powers*) and the methods required to objects playing the role (*requirements*). If an object of a class offering the requirements, plays the role, it is empowered with new methods. The definition of a role using the keyword `role` is similar to the definition of an interface; it is the specification of the powers acquired by the role in the form of abstract methods signatures. The only difference is that the role definition by means of the keyword `playedby` refers also to another interface, that in turn specifies the requirements which an object playing the role must satisfy.

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

3.2 Institutions and role implementation

In [0, 0, 0] roles are always associated with an instance of, and are definitionally dependent on, an institution. Roles add powers to objects playing the roles. Power means the possibility to modify also the state of the institution which defines the role and the state of the other roles defined in

```
class School {
  private int[][] marks;
  private Teacher[] teachers;
  private String schoolName;
  School (String schoolName) {
    this.schoolName = schoolName; ... }

  class StudentImpl realizes Student {
    private int studentID;
    public int getStudentID() { return studentID; }
    public void takeExam(int examCode; HomeWork hwk) {
      marks[studentID][examCode] =
        teachers[examCode].evalHomeWork(hwk);
    }
    public String getName() {
      return that.getName() + ", student at " + schoolName;
    }
  }

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

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

Figure 2: Institution and role implementation.

the same institution. In our running example, we have that the method for taking an exam in the school must be able to modify the private state of the school. For example, if the exam is successful, the grade should be added to the registry of exams in the school by the teacher. Analogously, the student's method for taking an exam can invoke the teacher's method of evaluating an examination. Powers, thus, seems to violate the standard encapsulation principle, where the private variables are visible to the class they belong to only. However, here, the encapsulation principle is preserved: all roles of an institution depend on the definition of the institution; so it is the institution itself which gives to the roles access to its private fields and methods. Since it is the institution itself which defines 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 is achieved in Java by means of the inner class construct. Thus, we extend the notion of inner class to allow roles to be implemented inside an institution (the outer class). The inner class construct is extended with the keyword `realizes` which specifies the name of the role specification, the inner class is implementing. An institution is simply a class with an inner class realizing roles in the very same way as a class implements an interface. In Figure , `StudentImpl` (`TeacherImpl`) realizes the role `Student` (`Teacher`), inside the institution `School`. Note that, a role (implementation) could itself be an institution with its own role implementations, it could enact other roles and, analogously, an institution could play a role. Moreover, roles can be implemented in different ways in the same institution.

Since the behavior of a role instance depends on its player, 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 implementation is invoked. The referred object has the type defined by the role requirements or a subtype. We do not need a special expression for creating instances of the inner classes implementing roles, because we use the Java inner classes syntax: starting from an institution instance (or from a class name in case of static inner classes), the keyword `new` allows the creation of an instance of the role as an instance of the inner class, e.g., `harvard.new StudentImpl(chris)` in Figure . Note that all the constructors of role implementations have at least a (implicit) parameter which must be bound to the player of the role and become the value of `that`.

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

3.3 Playing a role

A role represents a perspective on an object. An object has different (or additional) properties when it is seen in the perspective of a certain role, and it can perform new activities, which we call powers, as specified by the role definition. In Steimann [0]'s terminology, a role is a type specifying behavior.

When an object is seen under the perspective of a role, we want that the object has a specific state for it. This state is different from the player's one, it is specific to each role in each institution, and it can evolve with time by invoking methods on the roles (or on other roles of the same institution as we have seen in the running example). This state is given by a role instance which is associated with the player. Since a role represents the perspective on an object, the object playing the role should be able to invoke the role's methods 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 a person is a student and a student can be asked to return its enrollment number, we want to be able to invoke the method on the person as a student without referring to the student role instance.

The same methods will have a different behavior according to the role which the object plays when they are invoked. On the other hand, methods of a role can exhibit different behaviors according to whom is playing it. So a method of student returning the name of the student together with the name of the school returns different values for the name according to whom is playing the role of student. This is possible since the implementation of methods representing

```
class TestRole {
  public static void main(String[] args) {
    Person  chris = new Person("Christine", 1234);
    Person  george =
      new QualifiedPerson("George", 5678, 9876);
    School  harvard = new School("Harvard");
    School  mit = new School("MIT");
    harvard.new StudentImpl(chris);
    harvard.new TeacherImpl(george);
    mit.new TeacherImpl(george);
    String x = ((harvard.StudentImpl) chris).getName();
    String y = ((harvard.TeacherImpl) george).getName();
    String z =
      ((Teacher)(mit.TeacherImpl) george).getName();
    ((harvard.StudentImpl) chris).takeExam(...,...);
  }
}
```

Figure 3: Playing a role.

powers uses the methods required by the role to its player in order to play the role. These required methods obviously can access the state of the player since they are part of the implementation of the player.

Roles are always roles in an institution. Hence, an object can play at the same moment the same role more than once, albeit in different institutions. Instead, we do not consider the case of an object playing the *same role* (same implementation) 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 implementation, both the object and the institution where it plays the role must be specified. We call this methodology *role casting.* Type casting in Java allows to see the same object under different perspectives while maintaining the same structure and state. In contrast, role casting views an object as having a different state and different behaviors when playing different roles. So, the last syntactic change in powerJava is the introduction of *role casting expressions* extending the original Java syntax for casting. A role cast specifies both the role and the instance of the institution the role belongs to. For example, in `(harvard.TeacherImpl) george`, in Figure , the person `george` is casted to its role `harvard.TeacherImpl` of type `School.TeacherImpl`. It is important to observe that role casting is done to the inner class implementing the role but the role instance can always be type casted to the role as well as it can be done with Java interfaces: `((Teacher)(harvard.TeacherImpl) george).getName()`. While in the previous case it was possible to use all the methods of the specific implementation, in this case, only the methods that are specified in the role definition can be applied.

4. CONCLUSIONS AND RELATED WORK

In this paper we introduce a new view on roles in OO programming languages based on an ontological analysis of the notion of role. We introduce this model of roles in an extention of Java, called powerJava. Details about the implementation can be found in [0] and the precompiler is available at `http://www.powerjava.org`. powerJava is also been used as a coordination language dealing with concurrency [0].

Many works on the introduction of roles in programming languages [0, 0, 0, 0] consider roles as dynamic specializations of classes, e.g., a customer is seen as a specialization of the class person. This methodology does not capture the fact that a role like customer can be played both by a person and by an organization (that is not a person). Roles as specializations prevent realizing that a role is always associated not only with a player, but also to an institution, which defines it. This intuition sometimes emerges also in these frameworks: in [0] the authors say "a role is *visible* only within the scope of the specific application that created it", but context are not first class citizens like institutions are in our model.

Some other works adopt a closer methodology: roles are seen as instances which are associated with objects. Wong *et al.* [0] introduce a parallel role class hierarchy connected by a "played-by" relationship to the object class hierarchy. However, they fail to capture the intuition that a role depends on the context defining it. Moreover, the method lookup as delegation they adopt 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. This is not a desired feature in a language like Java.

In [0] it is recognized that a role depends on its player and that the properties of the role are present only due to the perspective the role is seen from. However, 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 conflits with our position, which we adopt from Steimann [0], of roles as interfaces: roles are partial descriptions of behavior, they shadow the other properties of their players, rather than inheriting them.

Our approach share the idea of gathering roles inside wider entities with languages like Object Teams [0] and Caesar [0]. However, these languages emerge as refinements of *aspect oriented* languages aiming at resolving some of their practical limitations. Aspects fit our conceptual model as well: e.g., when the execution of methods gives raise, by advice weaving, to the execution of a method of a role, in our model this means that the actions of an object playing a role "count as" actions executed by the role itself. Finally, our notion of role, as a double-sided interface, bears some similarities with Traits [0] and Mixins. However, they are different as, with a few exceptions, e.g., [0], they are not used to extend instances, like roles do, but classes.

5. REFERENCES

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

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

[3] M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct: Introducing powerJava. In *Procs. of MTCoord'05 at Coordination'05*, 2005.

[4] M. Baldoni, G. Boella, and L. van der Torre. Social roles, from agents back to objects. In *Proc. of WOA 2005: Dagli oggetti agli agenti, simulazione e analisi formale di sistemi complessi*, Camerino, Italy, november 2005. Pitagora Editrice Bologna.

[5] L. Bettini, V. Bono, and S. Likavec. A core calculus of mixin-based incomplete objects. In *Procs. of FOOL Workshop*, pages 29–41, 2004.

[6] G. Boella and L. van der Torre. An agent oriented ontology of social reality. In *Procs. of FOIS'04*, pages 199–209, Torino, 2004. IOS Press.

[7] G. Boella and L. van der Torre. Attributing mental attitudes to roles: The agent metaphor applied to organizational design. In *Procs. of ICEC'04*. IEEE Press, 2004.

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

[9] M. Dahchour, A. Pirotte, and E. Zimanyi. A generic role model for dynamic objects. In *Procs. of CAiSE'02*, volume 2348 of *LNCS*, pages 643–658. Springer, 2002.

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

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

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

[13] 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.

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

[15] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Procs. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100. ACM Press, 2004.

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

[17] N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In Springer Verlag, editor, *LNCS, vol. 2743: Procs. of ECOOP'03*, pages 248–274, Berlin, 2003.

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

[19] F. Steimann. A radical revision of UML's role concept. In *Procs. of UML2000*, pages 194–209, 2000.

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

[21] F. Steimann, W. Siberski, and T. Kühne. Towards the systematic use of interface in java programming. In *Proc. of 2nd Int. Conf. on the Principle and Practice of Programming in Java*, pages 13–17, 2003.

[22] 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.