

Interaction among Objects via Roles

Sessions and Affordances in Java

Matteo Baldoni
Dipartimento di Informatica
Università di Torino - Italy
baldoni@di.unito.it

Guido Boella
Dipartimento di Informatica
Università di Torino - Italy
guido@di.unito.it

Leendert van der Torre
University of Luxembourg
Luxembourg
leendert@vandertorre.com

ABSTRACT

In this paper we present a new vision in object oriented programming languages where the objects' attributes and operations depend on who is interacting with them. This vision is based on a new definition of the notion of role, which is inspired to the concept of affordance as developed in cognitive science. The current vision of objects considers attributes and operations as being objective and independent from the interaction. In contrast, in our model interaction with an object always passes through a role played by another object manipulating it. The advantage is that roles allow to define operations whose behavior changes depending on the role and the requirements it imposes, and to define session aware interaction, where the role maintains the state of the interaction with an object. Finally, we discuss how roles as affordances can be introduced in Java, building on 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 is that the attributes and operations of an object should be associated with it. The interaction with the object is made via its public attributes and via its public operations. The implementation of an operation is specific of the class and can access the private state of it. This allows 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 from the other objects manipulating it; thus, the implementation can be changed without changing the interaction capabilities of the object.

This view can be likened with the way we interact with objects in the world: the same operation of switching a device on is implemented in different manners inside different kinds of devices, depending on their functioning. The philosophy behind object orientation, however, views reality in a naive way. It rests on the assumption that the attributes and operations of objects are objective, in the sense that they are the same whatever is the object interacting with them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ 2006, August 30 – September 1, 2006, Mannheim, Germany.
Copyright 2006 ACM . . . \$5.00.

This view limits sometime the usefulness of object orientation:

1. Every object can access all the public attributes and invoke all the public operations of every other object. Hence, it is not possible to distinguish which attributes and operations are visible for which classes of interacting objects.
2. The object invoking an operation (caller) of another object (callee) is not taken into account for the execution of the method associated with the operation. Hence, when an operation is invoked it has the same meaning whatever the caller's class is.
3. The values of the private and public attributes of an object are the same for all other objects interacting with it. Hence, the object has always only one state.
4. The interaction with an object is session-less since the invocation of an operation does not depend on the caller. Hence, the value of private and public attributes and, consequently, the meaning of operations cannot depend on the preceding interactions with the object.

The first three limitations hinder modularity, since it would be useful to keep distinct the core behavior of an object from the different interaction possibilities that 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 of each method. The last limitation complicates the modelling of distributed scenarios where communication follows protocols.

Programming languages like Fickle [9] address the second and third problem by means of dynamic reclassification: an object can change class dynamically, and its operations change their meaning accordingly. However, Fickle does not represent the dependence of attributes and operations from the interaction. Aspect programming focuses too on the second and third issue, while it is less clear how it addresses the other ones.

Sessions are considered with more attention in the agent oriented paradigm, which bases communication on protocols ([10, 13]). A protocol is the specification of the possible sequences of messages exchanged between two agents. Since not all sequences of messages are legal, the state of the interaction between two agents must be maintained in a session. Moreover, not all agents can interact with other ones using whatever protocol. Rather the interaction is

allowed only by agents playing certain roles. However, the notion of role in multi-agents systems is rarely related with the notion of session of interaction. Moreover, it is often related with the notion of organization rather than with the notion of interaction.

Roles in object oriented programming, instead, aim at modelling the properties and behaviors of entities which evolve over time, while the interaction among objects is mostly disregarded [7, 14, 18]. Hence, they adopt the opposite perspective.

In this paper, we address the four problems above in object oriented programming languages by using a new notion of role we introduced in [4]. This is inspired to research in cognitive science, where the naive vision of objects is overcome by the so called ecological view of interaction in the environment. In this view, the properties (attributes and operations) of an object are not independent from whom is interacting with it. An object “affords” different ways of interaction to different kinds of objects.

The structure of this paper is as follows. In Section 2 we discuss the cognitive foundations of our view of objects. In Section 3 we define roles in terms of affordances. In Section 4 we show how our approach to roles impact on the design of a new object oriented programming language, powerJava. Related work and conclusion end the paper.

2. ROLES AS AFFORDANCES

The naive view of objects assigns them objective attributes and operations which are independent from the observer or other objects interacting with them. Instead, recent developments in cognitive science show that attributes and operations, called *affordances*, emerge only at the moment of the interaction and change according to what kind of object is interacting with another one.

The notion of affordance has been developed by a cognitive scientist, James Gibson, in a completely different context, the one of visual perception [11] (p. 127):

“The affordances of the environment are what it offers the animal, what it provides or furnishes, either for good or ill. The verb to afford is found in the dictionary, but the noun affordance is not. I have made it up. I mean by it something that refers to both the environment and the animal in a way that no existing term does. It implies the complementarity of the animal and the environment... If a terrestrial surface is nearly horizontal (instead of slanted), nearly flat (instead of convex or concave), and sufficiently extended (relative to the size of the animal) and if its substance is rigid (relative to the weight of the animal), then the surface affords support... Note that the four properties listed - horizontal, flat, extended, and rigid - would be physical properties of a surface if they were measured with the scales and standard units used in physics. As an affordance of support for a species of animal, however, they have to be measured relative to the animal. They are unique for that animal. They are not just abstract physical properties.

The same layout will have different affordances for different animals, of course, insofar as each animal has a different repertory of acts. Different animals will perceive different sets of affordances therefore.”

Gibson refers to an ecological perspective, where animals and the environment are complementary. But the same vision can be transferred to objects. By “environment” we intend a set of objects and by “animal of a given specie” we intend another object of a given class which manipulates them. Besides physical objective properties objects have affordances when they are considered relative to an object managing them. Thus, we have that the properties which characterize an object in the environment depend on the properties of the interactant. Thus, the interaction possibilities of an object in the environment depend on the properties of the object manipulating it.

How can we use this vision to introduce new modelling concepts in object oriented programming? Different sets of affordances of an object are associated with each different way of interaction with the objects of a given class. We will call a *role type* the different sets of affordances of an object. A role type represents the interaction possibilities which depend on the class of the interactant manipulating the object: the *player* of the role. To manipulate an object the caller of a method has to specify the role in which the interaction is made. But an ecological perspective cannot be satisfied by considering only occasional interactions between objects. Rather it should also be possible to consider the continuity of the interaction for each object, i.e., the state of the interaction. In terms of a distributed scenario, a session. Thus a given role type can be instantiated, depending on a certain player of a role (which must have the required properties), and the *role instance* represents the state of the interaction with that role player.

3. ROLES AND SESSIONS

The idea behind affordances is that the interaction with an object does not happen directly with it by accessing its public attributes and invoking its public operations. Rather, the interaction with an object happens via a role: to invoke an operation, it is necessary first to be the player of a role offered by the object the operation belongs to, and second to specify in which role the method is invoked. The roles which can be played depend on the properties of the player of the role (the *requirements*), since the affordances depend on the “different repertories of acts”.

Thus, a class is seen as a cluster of classes gathered around a central class. The central class represents the core state and behavior of the object. The other classes, the role types, are the containers of the operations specific of the interaction with a given class, and of the attributes characterizing the state of the interaction. Not only the kind of attributes and methods depend on the class of the interacting object and on the role in which it is interacting, but also the values of these attributes may vary according to a specific interactant: they are memorized in a role instance. A role instance, thus, models the session of the interaction between two objects and can be used for defining protocols.

Since a role represents the possibilities offered by an object to interact with it, the methods of a role must be able to affect the core state of the objects they are roles of and to access their operations; otherwise, no effect could be made by the player of the role on the object the role belongs to. So a role, even if it can be modelled as an object, is, instead different: a role depends both on its player and on the object the role belongs to and it can access the state of the object the role belongs to.

Many objects can play the same role as well as the same object can play different roles. In Figure 1 we depict the different possibil-

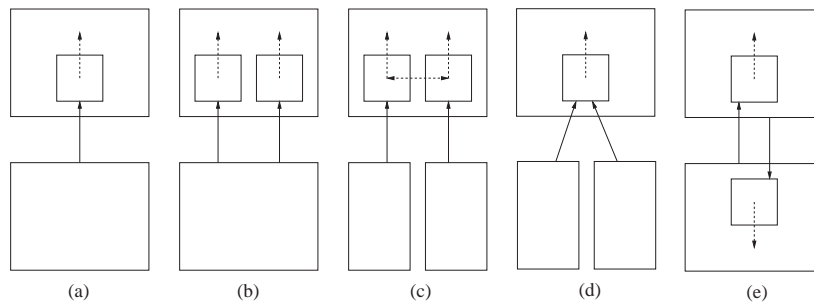


Figure 1: The possible uses of roles as affordances.

ities. *Boxes* represent objects and role instances (included in external boxes). *Arrows* represent the relations between players and their roles, *dashed arrows* the access relation between objects.

- Drawing (a) illustrates the situation where an object interacts with another one by means of the role offered by it.
- Drawing (b) illustrates an object interacting in two different roles with another one. This situation is used when an object implements two different interfaces for interacting with it, which have methods with the same signature but with different meanings. In our model the methods of the interfaces are implemented in the roles offered by the object to interact with it. Moreover, the two role instances represent the two different states of the two interactions between the two objects.
- Drawing (c) illustrates the case of two objects which interact with each other by means of the two roles of another object (the two role instances may be of the same class). This object can be considered as the context of interaction. This achieves the separation of concerns between the core behavior of an object and the interaction possibilities in a given context. The meaning of this scenario for coordination has been discussed in [5]; in this paper, we used as a running example the well-known 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. The coordinated objects are the players of the role chopstick and philosopher. The former role is played by objects which produce information, the latter by objects which consume them. None of the players contains the code necessary to coordinate with the others, which is supplied by the roles.
- In drawing (d) a degenerated but still useful situation is depicted: a role does not represent the individual state of the interaction with an object, but the collective state of the interaction of two (or more) objects playing the same role instance. This scenario is useful when it is not necessary to have a session for each interaction.
- In drawing (e) two objects interact with each other, each one 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* [3]. The symmetry of roles is closer to the traditional vision of roles as ends of a relation.

4. AFFORDANCES IN POWERJAVA

Baldoni *et al.* [3] introduce roles as affordances in powerJava, an extension of the object oriented programming language Java. Java is extended with:

1. A construct defining the role with its name, the requirements and the signatures of the operations which represent the affordances of the interaction with an object by playing the role.
2. The implementation of a role, inside an object and according to its definition.
3. A construct for playing a role and invoking the operations of the role.

We illustrate powerJava by means of an example. Let us suppose to have a printer which supplies two different ways of accessing to it: one as a normal user, and the other as a superuser. Normal users can print their jobs and the number of printable pages is limited to a given maximum. Superusers can print any number of pages and can query for the total number of prints done so far. In order to be a user one must have an account, which is printed on the pages. The role specification for the user and superuser is the following:

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

role SuperUser playedby AccountedPerson {
    int print(Job job);
    int getTotalPages();}
```

Requirements must be implemented by the objects which act as players.

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

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

Instead, affordances are implemented in the class in which the role itself is defined. To implement roles inside it we revise the notion of *Java inner class*, by introducing the new keyword `definerole` instead of `class` followed the name of the role definition that the class is implementing (see the class `Printer` in Figure 2). Role specifications cannot be implemented in different ways in the same class and we do not consider the possibility of extending role implementations (which is, instead, possible with inner classes), see [4] for a deeper discussion.

As a Java inner class, a role implementation has access to the private fields and methods of the outer class (in the above example the private method `print` of `Printer` used both in role `User` and in role `SuperUser`) and of the other roles defined in the outer class. This possibility does not disrupt the encapsulation principle since all roles of a class are defined by the same programmer who defines the class itself. In other words, an object that has assumed a given role, by means of the role's methods, has access and can change the state of the object the role belongs to and of the sibling roles. In this way, we realize the affordances envisaged by our analysis of the notion of role.

The class implementing the role is instantiated by passing to the constructor an instance of an object satisfying the requirements. The behavior of a role instance depends on the player instance of the role, so in the method implementation the player instance can be retrieved via a new reserved keyword: `that`, which is used only in the role implementation. In the example of Figure 2 `that.getLogin()` is parameter of the method `print`.

All the constructors of all roles have an implicit first parameter which must be passed as value the player of the role. The reason is that to construct a role we need both the object the role belongs to (the object the construct `new` is invoked on) and the player of the role (the first implicit parameter). This parameter has as its type the requirements of the role and it is assigned to the keyword `that`. A role instance is created by means of the construct `new` and by specifying the name of the "inner class" implementing the role which we want to instantiate. This is like it is done in Java for inner class instance creation. Differently than other objects, role instances do not exist by themselves and are always associated to their players and to the object the role belongs to.

The following instructions create a printer object `laser1` and two person objects, `chris` and `sergio`. `chris` is a normal user while `sergio` is a superuser. Indeed, instructions four and five define the roles of these two objects w.r.t. the created printer.

```
Printer hp8100 = new Printer();
//players are created
Person chris = new Person();
Person sergio = new Person();
//roles are created
hp8100.new User(chris);
hp8100.new SuperUser(sergio);
```

An object has different (or additional) properties when it plays a certain role, and it can perform new activities, as specified by the role definition. Moreover, a role represents a specific state which is different from the player's one, which can evolve with time by invoking methods on the roles. The relation between the object and the role must be transparent to the programmer: it is the object which has to maintain a reference to its roles.

Methods can be invoked from the players, given that the player is seen in its role. To do this, we introduce the new construct:

```
receiver <-(role) sender
```

This operation allows the `sender` (the player of the role) to use the affordances given by `role` when it interacts with the `receiver` the role belongs to. It is similar to *role cast* as introduced in [1, 4, 5] but it stresses more strongly the interaction aspect of the two involved objects: the `sender` uses the role defined by the `receiver` for interacting with it. Let us see how to use this construct in our running example.

In the example the two users invoke method `print` on `hp8100`. They can do this because they have been empowered of printing by their roles. The act of printing is carried on by the private method `print`. Nevertheless, the two roles of `User` and `SuperUser` offer two different way to interact with it: `User` counts the printed pages and allows a user to print a job if the number of pages printed so far is less than a given maximum; `SuperUser` does not have such a limitation. Moreover, `SuperUser` is empowered also for viewing the total number of printed pages. Notice that the page counter is maintained in the role state and persists through different calls to methods performed by a same `sender/player` towards the same `receiver` as long as it plays the role.

```
(hp8100 <-(User) chris).print(job1);
(hp8100 <-(SuperUser) sergio).print(job2);
(hp8100 <-(User) chris).print(job3);
System.out.println("Chris printed "+
    (hp8100 <-(User) chris).getPrintedPages());
System.out.println("The printer printed" +
    (hp8100 <-(SuperUser) sergio).getTotalPages());
```

By maintaining a state, a role can be seen as realizing a *session-aware interaction* between objects, in a way that is analogous to what done by cookies on the WWW or Java sessions for JSP and Servlet. So in our example, it is possible to visualize the number of currently printed pages, as in the above example. Note that, when we talk about playing a role we always mean playing a role instance (or *qua individual* [17] or *role enacting agent* [8]) which maintains the properties of the role.

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 it is invoked. It is sufficient to specify which is the role of a given object, we are referring to. In the example `chris` can become also `SuperUser` of `hp8100`, besides being a normal user

```
hp8100.new SuperUser(chris);
(hp8100 <-(SuperUser) chris).print(job4);
(hp8100 <-(User) chris).print(job5);
```

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

```

class Printer {
  private int totalPrintedPages = 0;
  private void print(Job job, Login login) {
    totalPrintedPages += job.getNumberPages(); ... // performs printing
  }
  definerole User {
    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 {
    public int print(Job job) {
      Printer.this.print(job, that.getLogin());
      return totalPrintedPages;
    }
    public int getTotalpages() { return totalPrintedPages; }
  }
}

```

Figure 2: The Printer class and its affordances

5. RELATED WORK

There is a huge amount of literature concerning roles in programming languages, knowledge representation, multiagent systems and databases. Thus we can compare our approach only with a limited number of other approaches.

First of all, our approach is consistent with the definition of roles in ontologies given by Masolo *et al.* [17], as we discuss in [4].

The term of role is already used also in Object Oriented modelling languages like UML and it is related to the notion of collaboration: “while a classifier is a complete description of instances, a classifier role is a description of the features required in a particular collaboration, i.e. a classifier role is a projection of, or a view of, a classifier.” This notion has several problems, thus Steimann [19] proposes a revision of this concept merging it with the notion of interface. However, by role we mean something different from what is called role in UML. UML is inspired by the relation view of roles: roles come always within a relation. In this view, which is also shared by, e.g., [15, 16], roles come in pairs: buyer-seller, client-server, employer-employee, *etc.*. In contrast, we show, first, that the notion of role is more basic and involves the interaction of one object with another one using one single role, rather than an association. Second, we highlight that roles have a state and add properties to their players besides requiring the conformance to an interface which shadows the properties of the player.

A leading approach to roles in programming languages is the one of Kristensen and Osterbye [14]. A role of an object is “a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects”. Even if at first sight this definition seems related to ours, it is the opposite of our approach. By “a role of an object” they mean the role played by an object, we mean, instead, the role offered by an object to interact with it. They say a role is an integral part of the object and at the same time other objects need to see the object in a certain restricted way by means of roles. A person can have the role of bank employee, and thus its properties are extended with the properties of employee. In our approach, instead, by a role of an object we mean the role offered by an object to interact with it by playing the role: roles allow objects which can interact in different ways with play-

ers of different roles. We focus on the fact that to interact with a bank an object must play a role defined by the bank, e.g., employee, and to play a role some requirements must be satisfied.

Roles based on inner classes have been proposed also by [12, 20]. However, their aim is to model the interaction among different objects in a context, where the objects interact only via the roles they play. This was the original view of our approach [1], too. But in this paper and in [3] we extend our approach to the case of roles used to interact with a single object to express the fact that the interaction possibilities change according to the properties of the interactants.

Aspect programming addresses some of the concerns we listed in the Introduction. In particular, aspect weaving allows to change the meaning of methods. Sometimes aspects are packed into classes to form roles which have their own state [12]. However, the aim of our proposal is different from modelling crosscutting concerns. Our aim is to model the interaction possibilities of an object by offering different sets of methods (with a corresponding state) to different objects by means of roles. Roles are explicitly expressed in the method call and specify the affordances of an object: it is not only the meaning of methods which changes but 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. In this way, not only the interaction possibilities offered by an object would depend on the caller of a method, but they would also depend on the environment where the interaction happens. Consider the `within` construct in Object Teams/Java [12] 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, the effect of the environment can be better modelled as a crosscutting concern. Thus aspect programming is a complementary approach with respect to our one.

Some patterns partially address the same problems of this paper. For example, the strategy design pattern allows 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.

Baumer *et al.* [6] propose the role object pattern to solve the problem of providing context specific views of the key abstractions of a system. They argue that different context-specific views cannot be integrated in the same class, otherwise the class would have a bloated interface, and unanticipated changes would result in recompilations. Moreover, it is not possible either to consider two views on an object as an object belonging to two different classes, or else the object would not have a single identity. 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. Our proposal is distinguished by the fact that roles are always roles of an institution. As a consequence they do not consider the additional methods of the roles as powers which are implemented using also the requirements of the role. Finally, in their model, since the role and its player share the same interface, it is not possible to express roles as partial views on the player object.

6. CONCLUSION

In this paper we introduce the notion of affordance developed in cognitive science to extend the notion of object in the object orientation paradigm. In our model objects have attributes and operations which depend on the interaction with other objects, according to their properties. Sets of affordances form role types whose instances are associated with players which satisfy the requirements associated with roles. Since role instances have attributes they provide the state of the interaction with an object.

In [1] we present a different albeit related notion of role, with a different aim: representing the organizational structure of institutions which is composed of roles. The organization represents the context where objects interact only via the roles they play by means of the powers offered by their roles (what we call here affordances). E.g., a class representing a university offers the roles of student and professor. The role student offers the power of giving exams to players enrolled in the university. In [5] we explain how roles can be used for coordination purposes. In [4] we investigate the ontological foundations of roles. In [2] we describe the preprocessor translating powerJava into Java. In this paper, instead, we use roles to articulate the possibility of interaction provided by an object.

Future work concerns modelling the symmetry of roles. In particular, the last diagram of Figure 1 deserves more attention. For example, the requirements to play a role must include the fact that the player must offer the symmetric role (e.g., initiator and participant in a negotiation). Moreover, in that diagram the two roles are independent, while they should be related. Finally, the fact that the two roles are part of a same process (e.g., a negotiation) should be represented, in the same way we represent that student and professor are part of the same institution.

7. REFERENCES

- [1] M. Baldoni, G. Boella, and L. van der Torre. Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *LNCS 3862: Procs. of PROMAS'05 workshop at AAMAS'05*, pages 57–75, Berlin, 2005. Springer.
- [2] M. Baldoni, G. Boella, and L. van der Torre. Social roles, from agents back to objects. In *Procs. of WOA'05 Workshop*, Bologna, 2005. Pitagora.
- [3] 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.
- [4] M. Baldoni, G. Boella, and L. van der Torre. Powerjava: ontologically founded roles in object oriented programming language. In *Procs. of OOPs Track of ACM SAC'06*, pages 1414–1418. ACM, 2006.
- [5] 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.
- [6] D. Baumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern. In *Procs. of PLOP'02*, 2002.
- [7] J. Cabot and R. Raventos. Conceptual modelling patterns for roles. In *LNCS 3870: Journal on Data Semantics V*, pages 158–184, Berlin, 2006. Springer.
- [8] M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *Procs. of AAMAS'03*, pages 489–496, New York (NJ), 2003. ACM Press.
- [9] 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.
- [10] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multiagent systems. In *LNCS 2935: Procs. of AOSE'03*, pages 214–230, Berlin, 2003. Springer.
- [11] J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, New Jersey, 1979.
- [12] S. Herrmann. Roles in a context. In *Procs. of AAI Fall Symposium Roles'05*. AAAI Press, 2005.
- [13] T. Juan, A. Pearce, and L. Sterling. Roadmap: extending the gaia methodology for complex open system. In *Procs. of AAMAS'04*, pages 3–10, 2002.
- [14] B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160, 1996.
- [15] F. Loebe. Abstract vs. social roles - a refined top-level ontological analysis. In *Procs. of AAI Fall Symposium Roles'05*, pages 93–100. AAAI Press, 2005.
- [16] C. Masolo, G. Guizzardi, L. Vieu, E. Bottazzi, and R. Ferrario. Relational roles and qua-individuals. In *Procs. of AAI Fall Symposium Roles'05*. AAAI Press, 2005.
- [17] 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.
- [18] M. Papazoglou and B. Kramer. A database model for object dynamics. *The VLDB Journal*, 6(2):73–96, 1997.
- [19] F. Steimann. A radical revision of UML's role concept. In *Procs. of UML2000*, pages 194–209, 2000.
- [20] T. Tamai. Evolvable programming based on collaboration-field and role model. In *Procs. of IWPSE'02*, pages 1–5. ACM, 2002.