Adding Roles to Relationship Patterns

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

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

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

***Abstract*—In this paper we study how roles can be added to patterns modelling relationships in Object Oriented programming, and which new relationship patterns can be introduced using roles. Relationships can be introduced in programming languages either by reducing them to attributes of the objects which participate in the relationship, or by modelling the relationship itself as a class whose instances have the participants of the relationships among their attributes. However, even if roles have been recognized as an essential component of relationships, also in modelling languages like UML, they have not been introduced in Object Oriented programming when it is necessary to model relationships. Introducing roles allows to add attributes and behaviors to the participants in the relationship, rather than to the relationship itself, and to distinguish natural types as classes participating in the relationships from the roles the participants acquire in the relationships. In this paper we show how the role model proposed in powerJava can be used to endow relationships with roles, both in the relationship as attribute and in the relationship object pattern. Finally, since these patterns have different advantages and limitations, we propose a third pattern based on roles which benefits from the advantages of the two previous patterns when modelling relationships.**

I. Introduction

The need of introducing the notion of relationship as a first class citizen in Object Oriented (OO) programming, in the same way as this notion is used in OO modelling, has been argued by several authors, at least since Rumbaugh [1]. For example, one would like to be able to model the following scenario: a student can be related to a university by an enrollment relationship, he can attend a course, and give exams. Moreover, a course can be a basic course in one curriculum and an advanced one in another. Similarly, other relationships link professors to students and courses, students to tutors, *etc*. Another example is the case of a contract net protocol, where two objects participate in a negotiation relationship, and inside this they can perform negotiation moves.

Relationships are also known as collaborations or associations, like they are called in UML, to distinguish them from specialized relationships like aggregation, relating an object to its parts, and inheritance, relating a class to a superclass.

Rumbaugh [1] claims that relationships are complementary to, and as important as, objects themselves. Thus, they should not only be present in modelling languages, like ER or UML, but they also should be available in programming languages, either as primitives, or, at least, represented by means of suitable patterns.

Two main alternatives have been proposed by Noble [2] for modelling relationships by means of patterns:

- The relationship as attribute pattern: the relationship is modelled by means of an attribute of the objects which participate in the relationship. For example, the `Attend` relationship between a `Student` and a `Course` can be modelled by means an attribute `attended` of the `Student` and of an attribute `attendee` of the `Course`.
- The relationship object pattern: the relationship is modelled as a third object linked to the participants. A class `Attend` must be created and its instances related to each pair of objects in the relationship. This solution underlies programming languages introducing primitives for relationships, e.g., Bierman and Wren [3].

These two solutions have different pros and cons, as Noble [2] discusses. But they both fail to capture an important modelling and practical issue. If we consider the kind of examples used in the works about the modelling of relationships, we notice that relationships are also essentially associated with another concept: students are related to tutors or professors [3], [4], basic courses and advanced courses [4], customers buy from sellers [5], employees are employed by employers, underwriters interact with reinsurers [2], *etc.* From the knowledge representation point of view, as noticed by ontologists like Guarino and Welty [6], these concepts are not natural kinds like person or organization. Rather, they all are *roles* involved in a relationship.

Roles have different properties than natural kinds, and, thus, are difficult to model with classes: roles can be played by objects of different classes, they are dynamically acquired, they depend on other entities - the relationship they belong to and their players. Moreover, when an object of some natural type plays a certain role in a relationship, it acquires new properties and behaviors. For example, a student in a course has a tutor, he can give the exam and get a mark for the exam, another property which exists only as far as he is a student of that course.

Thus, roles cannot simply be modelled as subclasses or superclasses of natural types by means of dynamic reclassification [7]: a student is not simply a subtype of person nor viceversa.

As Steimann [7] argues, there is an intrinsic role of roles as intermediaries between relationships and the objects that engage in them. Thus, in this paper, we focus on the following

research questions: How to introduce roles in the relationship as attribute pattern and in the relationship object pattern? Which other patterns are possible for modelling relationships when roles are introduced in Object Orientation? As subquestions: How to distinguish natural types from roles when designing a program? Which are the pros and cons of the two patterns when roles are added? How to overcome the limitations of the existing patterns?

In this paper we do not propose a new primitive of relationship in programming languages, but we introduce roles in patterns for relationships, and as methodology we use our model of roles in OO programming languages, an extension which adds roles to the Java programming language, called powerJava, described in [8], [9], [10], [11], for which a precompiler has been built.

The language powerJava introduces roles as a way to structure the interaction of an object with other objects calling their methods. Roles express the possibilities of interaction offered by the object to other ones, i.e., the methods they can call. First, these possibilities change according to the class of the callers of the methods. Second, a role maintains the state of the interaction with a certain individual caller. As roles have a state and a behavior, they share some properties with classes. However, roles can be dynamically acquired and released by an object playing them. Moreover, they can be played by different types of classes. This is why roles in powerJava can be useful in modelling relationships, where the behavior of an object changes when it enters a relationship, until it subsequently abandons it.

In Section II we discuss why and how relationships are introduced in OO programming. In Section III we discuss the link between relationships and roles. In Section IV we summarize our model of roles in powerJava and in Section V we use it to introduce roles in the relationship as attribute and relationship object patterns. In Section VI, we describe a new pattern combining the previous ones. Conclusions end the paper.

II. Introducing relations in OO

To understand the importance of relations in programming consider the efforts done to model relationships in defining patterns for them [2], [5], [12] or in extending existing languages like Java [3].

There is not yet a standard language with the relationship primitive, notwithstanding some interesting proposals like [3]. Hence, in this paper, to discuss the role of roles in relationships, we will focus on patterns for modelling relationships. The most important patterns for modelling relationships are the relationship as attribute pattern and the relationship object pattern [2]. We will not consider here other solutions like the collection object, mutual friends and active value patterns.

We will describe these two alternatives with reference to a university domain. Consider a student who can attend different kind of courses: basic ones and advanced ones. The same course can be a basic one in the curriculum of a senior student and an advanced one for junior student. A student can give the exam of the basic course he is attending and it is possible to send a message to the student of the course. Finally, a course is associated with a tutor if it is taken as a basic course; the tutor, which is not present in advanced courses, can be different for every student attending it.

The relationship as attribute pattern is described in Figure 1: the relationship between a student and a basic course he attends is modelled by means of an attribute `attends` of the instances of class `Student` which participate in the relationship. The type of the attribute is a set of `BasicCourse`. Symmetrically, the `Student` appears in the attribute `attendees` of the class `BasicCourse` of type set of `Student`. The `BasicCourse` is also related with other courses by a relationship representing the prerequisites.

This solution, however, does not allow to add a state and behavior to the elements related by the relationship. For example, it is not possible to specify a different tutor for each `Student` of the `BasicCourse`. Moreover, the `enrol` method is arbitrarily implemented in `BasicCourse` rather than in `Student`.

The relationship object pattern is instead described in Figure 2: the relationship `AttendBasicCourse` is modelled by a class whose instances link each `Student` to the `BasicCourse` he attends. The second solution solves some of the issues discussed in this section, in particular, it facilitates the cohesion of the program, by factoring in the class `AttendBasicCourse` all the relevant information. In particular, the class can contain the properties and the operations which the participants are endowed with when they enter the relationship. For example, a `Student` can take the exam of the `BasicCourse` and get a mark if he is successful. Note that the mark is a property belonging to the relationship. Moreover, the `Student` can be associated with a tutor in a `BasicCourse`.

Also this solution can be modelled in UML, which specifies information proper of an association via an association class, where the properties and behaviors of the relationship are represented. An association class has exactly one instance for each set of objects linked through the association and a lifetime delimited by the existence of the association. If a link is dissolved, the association class instance is destroyed. Due to the association, certain information exists that is specific to the association. In UML a dashed line is used to specify an association class.

But the relationship object solution shares with the relationship as attribute some limitations. First, we would like to model the university scenario introducing natural types like `Person` and `Course` rather than the `Student` and `BasicCourse` classes only. The reason for such modelling choice is that a `Person` is not always a `Student`, and he can play also other roles at the same time as he is a `Student`. Moreover, a `Person` is a `Student`, and, thus, he can give exams or receive communications concerning the course, only if he is related by the `AttendBasicCourse` relationship with a `Course` which he follows as a `BasicCourse`. He has different marks in different exams, and even different

```
class Student {
  String name;
  int number;
  HashSet<BasicCourse> attends; }

class BasicCourse {
  String code;
  String title;
  HashSet<Student> attendees;
  HashSet<BasicCourse> prerequisites;
  void enrol(Student s) {
    attendees.add(s);
    s.attends.add(this); } }
```

Fig. 1. The relationship as attribute pattern

```
class Student {
  String name;
  int number; }

class BasicCourse {
  String code;
  String title; }

class AttendBasicCourse {
  BasicCourse attended;
  Student attendee;
  Person tutor;
  int mark;
  AttendBasicCourse(Student s, Course c) {
    attended = c;
    attendee = s; }
  int giveExam(String work){mark = ...}
  void communicate(String text){...} }
```

Fig. 2. The relationship object pattern

students can have different tutors for the same course. Analogously a `Course` has a tutor only if it plays the role of `BasicCourse`.

Second, the relationship as attribute allows to add new properties and behaviors. However, it does not allow to satisfy completely the requirement that properties and behaviors are associated to the participants: this pattern does not distinguish which properties belong to the `Student` and which ones to the `BasicCourse`. This problem is more evident in the case of behaviors, since all the methods are invoked on the relationship object of class `AttendBasicCourse` rather than on the two related objects `Student` and `BasicCourse`. This is not only a modelling problem. It is not possible to have a method with the same name which should be called on either participant, `Student` and `BasicCourse`, with a different meaning. Thus, polymorphism is limited, for example, when the method should be specified as part of an interface implemented by both classes participanting in the relationship.

As noticed by Steimann [13], some of these problems cannot be solved by using subclassing: playing a role is not equivalent to subclassing (a `Person` becomes a `Student`), since a role can be played by instances of different classes. Consider the case of a role customer which can be played either by a person or by an organization.

Finally, these patterns do not consider a further dimension: the complexity of encapsulation when relations are considered. This problem has been highlighted by Noble and Grundy [5]: “Extra relationship objects existing ‘outside’ their participating objects may also be seen as breaking the participating object’s encapsulation [1]. The first point to note here is that many relationships occur between objects which are themselves parts of another aggregate object: that is, the relationship and the participating objects may all be encapsulated by another object. The second point here is that if encapsulation is broken by the relationship, this is because the encapsulated objects need to be accessed by the relationship object in order to implement the semantics of the relationship. Without the explicit relationship object, the analysis relationship would have to be implemented in another way, by being built in to the participating objects. If the relationship requires access to the ‘inside’ of an object breaking its encapsulation, these objects would therefore need to break each other’s encapsulation anyway. In short, using an explicit relationship object cannot worsen breaches of encapsulation. The root of the problem is not the relationship *object* (i.e., how the relationship is implemented), but the existence of the relationship as part of the problem domain.

In some circumstances, relationship objects may actually increase encapsulation, as the implementation of the relationship itself becomes encapsulated against the participating objects when it is moved in to a separate relationship object.”

In the next section we will explain how roles and relationships are related and how to overcome these problems.

III. Roles and relationships

Relations are deeply connected with roles. This is accepted in several areas: from modelling languages like UML and ER to knowledge representation discussed in ontologies and multiagent systems.

The link between roles and relationships is explicit in modelling languages like UML in the context of collaborations: a classifier role is a classifier like a class or interface, but “since the only requirement on conforming instances is that they must offer operations according to the classifier role, [...] they may be instances of any classifier meeting this requirement” [14]. In other words: a classifier role allows any object to fill its place in a collaboration no matter what class it is an instance of, if only this object conforms to what is required by the role. Classification by a classifier role is multiple since it does not depend on the (static) class of the instance classified, and dynamic (or transient) in the sense above: it takes place only when an instance assumes a role in a collaboration [15].

As noticed by Steimann [13], roles in UML are quite similar to the concept of interface, so that he proposes to unify the two concepts. Instead, there is more in roles than in interfaces. Steimann himself is aware of this fact: “another problem is that defining roles as interfaces does not cover everything one might expect from the role concept. For instance, in certain

situations it might be desirable that an object has a separate state for each role it plays, even for different occurrences in the same role. A person has a different salary and office phone number per job, but implementing the Employee interface only entails the existence of one state upon which behaviour depends. In these cases, modelling roles as adjunct instances would seem more appropriate."

To do this, Steimann [7] proposes to model roles as classifiers related to relationships, but such that these classifiers are not allowed to have instances. In Java terminology, roles should be modelled as abstract classes, where some behavior is specified, but not all the behavior, since some methods are left to be implemented in the class extending them. These abstract classes representing roles should be then extended by other classes. However, given that in Java multiple inheritance is not allowed, this solution is not viable, and roles can be identified to interfaces only.

Roles as defined in programming languages [11], [16], [17], instead, are different from interfaces, even if they share some properties with interfaces, like the fact of being partial specifications of behavior, thus allowing objects of different classes entering the same role in a relationship. In particular, roles have a state, add new operations to their players, and depend on a context [11], [17].

Also Whitehurst [18] argues that behavior depends on roles: "the behavior of an object can change depending on the role it plays. When an association is formed between two instances, the behavior of the associated instances is altered in some way. A real world example is a person who becomes a parent. The person has a parental association with a young person (a child) and the behavior of the person is changed due to this association".

Pearce and Noble [12] notice that relationships have similarities with roles. Objects in relationships have different properties and behavior: "behavioural aspects have not been considered. That is, the possibility that objects may behave differently when participating in a relationship from when they are not. Consider again the student-course example [...]. In practice, a course will have many more attributes, such as a curriculum, than we have shown. Such attributes will change over time in line with changes to the course. A useful constraint would be to prevent any changes when students are attending the course it would be unfair if the curriculum changed just before the exam! Thus, Course objects behave differently (i.e., they don't accept changes) when they are participating in a relationship from when they are not (i.e., they do accept changes)."

Thus, roles and relationships are strictly related.

In UML, it is possible to specify information and behavior specific to an association via an association class, but not with roles, which are partial description of behavior which do not add anything to their players.

In conclusion, it seems that besides the relationship objects it is necessary to introduce further objects representing the roles as adjunct instances of new classes.

IV. Roles in powerJava

Baldoni *et al.* [8], [9], [10], [11] introduce roles as affordances in powerJava, an extension of the object oriented programming language Java. powerJava is translated into Java by means of a precompiler, whose details are described in [11].

We only summarize here the powerJava language.

Java is extended with:

1) A construct defining the role with its name, the requirements and the signatures of the methods offered to the objects by playing the role, called powers.
2) The implementation of a role as a class, inside an object, and according to the definition of its powers.
3) A construct for playing a role and invoking the operations offered to the role.

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

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

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

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

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

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

Instead, roles are implemented in the class which offers the role. To implement roles inside it we revise the notion of Java *inner class*, by introducing the new keyword `definerole` instead of `class` followed by the name of the role definition that the class is implementing (see the class `Printer` in Figure 3).

As a Java inner class, the methods of a role implementation, called powers, have 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.

All the constructors of roles have an implicit first parameter to which it must be passed as value the player of the role: 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` starting from the object offering the role 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 laser1 = new Printer();

//players are created as Person
Person chris = new Person();
Person sergio = new Person();

//roles are created
laser1.new User(chris);
laser1.new SuperUser(sergio);
```

An object has different (or additional) properties when it plays a certain role, and it can perform new activities, the powers, 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 relationship 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.

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 Figure 3, `that.getLogin()` is a parameter of the method `print`.

Methods can be invoked from the players, given that the player is seen in its role. To do this, we introduce the new construct, called *role cast*. Role casting views an object as having a different state and different behaviors when playing different roles. Role casting allows to make transparent to the programmer the association of a role and an object instance: the programmer invokes a method of a role on the object playing it casted into the role; the language transforms this method invocation in a message sent to the delegated role instance, which is hidden in its player.

In the example the two users invoke the method `print` on `laser1`. They can do this because they have been empowered of printing by playing their roles. The act of printing is carried on by the private method `print`. Nevertheless, the two roles

```
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 Printer.this.totalPrintedPages;}
    public int getTotalpages()
     { return Printer.this.totalPrintedPages; }
  }
}
```

Fig. 3. The `Printer` class and its roles

of `User` and `SuperUser` offer two different ways to interact with the `Printer`: `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 of 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.

```
((laser1.User) chris).print(job1);
((laser1.SuperUser) sergio).print(job2);
((laser1.User) chris).print(job3);
System.out.println("The printer printed" +
  ((laser1.SuperUser) sergio).getTotalPages());
```

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 `laser1`, besides being a normal `user`

```
laser1.new SuperUser(chris);
((laser1.SuperUser) chris).print(job4);
((laser1.User)chris).print(job5);
```

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.

```
role Student playedby Person {
  int giveExam(String work); }

role BasicCourse playedby Course {
  void communicate(String text); }

class Person{
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended;

  definerole BasicCourse {
    Person tutor;
    void communicate (String text){
      Person.this.messages.add(text);}
    BasicCourse(Person t){
      tutor=t;
      Person.this.attended.add(this); }
  }
}

class Course {
  String code;
  String title;
  private HashTable registry =
     new HashTable();
  private HashSet<Student> attendees;
  private int evaluate(String x){...}

  definerole Student {
    int number;
    int mark;
    int giveExam(String work){
      mark = Course.this.evaluate(work);
      registry.set(that.hashCode(), mark);
      return mark;
      }
    Student (){
      Person.this.attended.add(this); }
  }
}
```

Fig. 4. Relationship-role as attribute pattern in powerJava

V. Relationships with roles using powerJava

In this section we describe how new patterns for modelling relationships with roles can be defined, in analogy with both the relationship as attribute and the relationship object pattern. We will use the example of Section II to present them.

First of all, using powerJava we can model instances of natural types like `Person` and `Course` which become, respectively, `Student` and `BasicCourse` when they enter the relationship. This is possible because `Student` and `BasicCourse` are roles represented in powerJava by instances associated with the players of the roles, which include the state and behaviors acquired by the players of the roles in the relationship.

In the relationship-role as attribute pattern, a relationship is not reduced only to two symmetric attributes `basicCourses` and `attendees`. The relationship is modelled also by means of a pair of roles. The `Person` plays the role of `Student` with respect to the `Course` and

```
public static void main (String[] args){
  Course c = new Course();
  Person p = new Person();
  //create a role Student for p in c
  Student s = c.new Student(p);
  BasicCourse b = p.new BasicCourse(c,tutor);
  //p as a Student of Course c gives the exam
  ((c.Student)p).giveExam(work);
  //a message text is sent
  ((p.BasicCourse)c).communicate(text); }
```

Fig. 5. Using the relationship-role as attribute in powerJava

the `Course` plays the role of `BasicCourse` with respect to the `Person` (see Figures 4, 5 and 6 where the UML representation is illustrated[1]).

Thus, the attribute `attendees` of type `Student` in `Course` is not replaced by one of type `Person`. Rather, `Student` is defined as role and `Person` is a class which can play the role (see the role definition connecting a role to the classes playing it). The role `Student` is associated with players of type `Person` in the role definition, which specifies that a `Student` can give an exam (`giveExam`). Analogously, the role `BasicCourse` is associated with players of type `Course` in the role definition, which specifies that a `Course` can communicate with the attendee.

The role `Student` is implemented locally in the class `Course` and, viceversa, the role `BasicCourse` is defined locally in the class `Person`. Note that this is not contradictory, since roles describe the way an object offers interaction to another one: a `Student` represents how a `Person` can interact with a `Course`, and, thus, the role is defined inside the class `Course`. Moreover the behavior associated with the role `Student`, i.e., giving exams, modifies the state of the class including the role (it access the `registry` variable) or calls its private methods (`evaluate`), thus violating the standard encapsulation. Analogously, the `communicate` method of `BasicCourse`, modifies the state of the `Person` hosting the role by adding a message to the queue. These methods, in powerJava terminology, exploit the full potentiality of powers of violating the standard encapsulation of objects.

To associate a `Person` and a `Course` in the relationship, the role instances must be created starting from the objects offering the role, e.g.: `c.new Student(p)` (see the `main` in Figure 5).

When the player of a role must invoke a power it must be first role casted to the role. For example, to invoke the method `giveExam` of `Student`, the `Person` must first become a `Student`. To do that, however, also the object offering the role must be specified, since the `Person` can play the role `Student` in different instances of `Course`; in this case the `Course c`: `((c.Student)p).giveExam(...)`.

The alternative relationship-role object pattern introduces an `AttendBasicCourse` class modelling the relationship between `Person` and `Course`. However, the

[1]The arrow starting from a crossed circle, in UML, represents the fact that the source class can be accessed by the arrow target class.

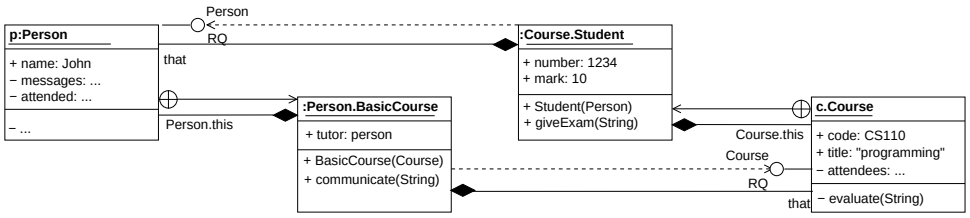

Fig. 6. The UML representation of the relationship-role as attribute pattern example

`AttendBasicCourse` class is not linked to a `Person` and a `Course`. Rather, the `Person` plays the role `Student` in the class `AttendBasicCourse` and the `Course` the role `BasicCourse` (see Figures 7 and 8). Like in the previous solution the roles are modelled as classes implemented, in this pattern, in the class `AttendBasicCourse` whose instances contain the properties and behaviors added when instances of `Person` and `Course`, respectively, participate in the relationship. Additionally, properties and behaviors which are associated to the relationship itself, like entering in the relationship and constraints on the participants can be added to the relationship class.

To relate a `Person` and a `Course` in a relationship, an instance of `AttendBasicCourse` must be created, together with an instance of `Student` played by the `Person` and of `BasicCourse` played by the `Course`. To invoke a power of `Student`, a `Person` must be role casted to the role `Student` starting from an instance of the class `AttendBasicCourse`.

With respect to the previous pattern, it is possible to notice that the roles can interact with each other: the role `Student` invokes in the method `giveExam` the private method `evaluate` of the `BasicCourse` role. However, the roles cannot anymore access the private state and methods of the player of the class. For this reason, it is necessary to add a public `getMessage` method in `Person`, and to define `evaluate` in the role rather than in the `Course` class.

The two patterns have different pros and cons; the following list integrates Noble [2]'s discussions on them.

Advantages of the Relationship-role as attribute pattern:

- It allows simple one-to-one relationships: it does not require a further class and its instance to represent the relationship between two objects.
- It allows to introduce a state and operations to the objects entering the relationship, which was not possible without roles in the relationship as attribute pattern.
- It allows the integration of the role and the element offering it by means of powers.
- It allows to show which roles can be offered by a class, and, thus, in which relationships they can participate, since they are all defined in the class.

Disadvantages of the Relationship-role as attribute pattern:

- It requires that the roles are already implemented offline inside the classes which participate in the relationship.
- It does not assure coherence of the pair of roles like student-course, buyer-seller, bidder-proponent, since they are defined separately in two different classes.
- The role cast to allow a player to invoke a power of its role requires to know the identity of the other participant in the relationship.
- It does not allow to distinguish which is the role played in the other object participating in the relationship (e.g., a `Student` in the `attendees` set of a `Course` can follow the `Course` as a `BasicCourse` or an `AdvancedCourse`).

Advantages of the Relationship-role object pattern:

- It allows to introduce a state and operations of the relationship besides the state and operations added to the objects entering the relationship.
- It allows to list all instances of the relationship and centralize operations like entering the relationship and to check constraints on the relationship.
- It enforces to create both role instances at the same time, since they are linked to the same relation instance, thus avoiding the risk of inconsistencies.
- It allows the integration of the role with the relationship and with the other role, since the powers of a role can access both. In this way it is possible to deal with coordination issues [8].
- To make a role cast it is necessary only to know the relationship instance, thus, the other participant can change without notice.
- It does not require that the classes of players already implement the role classes. To play a role it is sufficient to satisfy the requirements.

Disadvantages of the Relationship-role object pattern:

- It requires a further class and its instance.
- It does not allow the integration of roles with the objects offering them (e.g., `Student` is defined separately of the class `Course`, which, as a consequence, cannot be accessed). Thus, private variables of classes offering the role cannot be accessed anymore (see `registry` of `Course` in Figure 4), otherwise an object is required to offer additional public methods to access them (see `getMessage` in Figure 7), which endangers encapsulation.
- The roles cannot be tailored anymore with the class offering the role. E.g., the method `evaluate` cannot be anymore modelled as a private method of `Course`: different courses cannot evaluate an exam in different manners.

```
role Student playedby Person {
  int giveExam(String work); }

role BasicCourse playedby Course {
  void communicate(String text); }

class AttendBasicCourse{
  Student attendee;
  BasicCourse attended;
  static Hashset<AttendBasicCourse> all;

  definerole Student {
    int mark;
    int number;
    int giveExam(String work){
      mark = AttendBasicCourse.this.attended.evaluate(work); }
  }

  definerole BasicCourse {
    String program;
    Person tutor;
    private int evaluate(String work){...}
    void communicate(String text){
     //invoke the requirement of the Person playing the role
     AttendBasicCourse.this.attendee.that.getMessage(text);}
   }

   AttendBasicCourse(Person p, Course c, String p, Person t){
     attendee = this.new Student(p);
     attended = this.new BasicCourse(c);
     AttendBasicCourse.all.add(this);
   }
  void communicate(String text){
    foreach (AttendBasicCourse x: all)
      x.attended.communicate(text);}
  }

class Person{
  String name;
  private Queue messages;
  void getMessage(String text) {
    messages.add(text) };
}

class Course {
  String code;
  String title;
}

class University{
  public static void main (String[] args){
    Person p = new Person();
    Course c = new Course();
    a = new AttendBasicCourse(p,c,program,tutor);
    //p as a Student  gives the exam
    ((a.Student)p).giveExam(work);
    //c is used to send a message
    ((a.BasicCourse)c).communicate(text);}
```

Fig. 7. Relationship-role object pattern

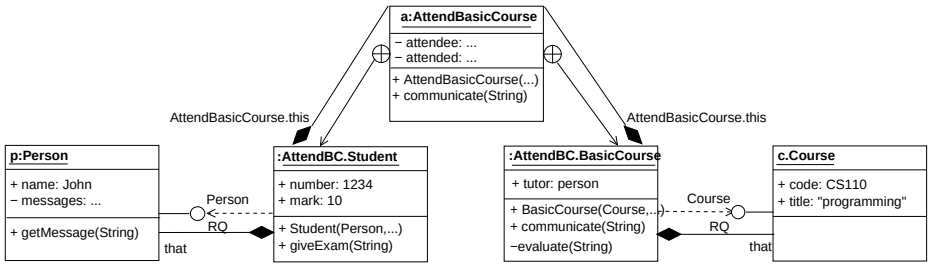

Fig. 8. The UML representation of the relationship-role object pattern example

In summary, we can define an informal program transformation, which is common to both patterns, to add roles to relationship patterns using powerJava:

1) Identify the natural types of the objects playing the roles (e.g., `Person` for `Student`, or `Person` and `Organization` for `Customer`).
2) Change the type of the classes which participate in the relationship from the name of the role to the name of the natural kind playing the role (now there can be more than one class playing the role); e.g., the class `Student` becomes `Person`.
3) Add a role definition relating the role to the natural types which can play the roles, or to an interface implemented by these natural types, and insert in the role definition the signature of the powers (e.g., `communicate`, `giveExam`).
4) Identify the two links to the participants in the relations, either in the classes representing the participants (e.g., `attendees` of type `Student` in `Course`), or in the class representing the relationship (for example `attendee` of type `Student` and `attended` of type `BasicCourse` in `AttendBasicCourse`).
5) In the same class the link belongs to, add a role class implementing the role definition with the same name as the type of the link (e.g., `Student` in the `BasicCourse` class which is now called `Course`, or `Student` and `BasicCourse` in `AttendBasicCourse`). Add to this role class the attributes and the implementation, according to the role definition, of the powers.
6) In the code which relates the two participant instances to the relationship, instead of adding the players to the links, first create two roles instances played by the respective playes, and, second, add these instances to the links modelling the relationship (either in the class of the players, e.g., `Person` or in the class modelling the relationship object, e.g., `AttendBasicCourse`).
7) When a method added by the relationship must be invoked, first, make a role cast from the object playing the role to the role it plays.

VI. A NEW RELATIONSHIP PATTERN WITH ROLES

The two relationship patterns added with roles still leave some unsolved problems. On the one hand, the relationship-role as attribute pattern allows a strict coupling between the role and the class offering it. For example, the role `Student` can access the class `Course` when the `Person` playing the role gives an exam: the registry of the courses can be added with a new entry with the mark of that exam.

On the other hand, the relationship-role object pattern allows a better coordination of the two roles. All roles of a relationship share the same namespace, and, thus, can access each other and the relation too. In this way, it is possible to define the interaction between the roles separately from the classes of possible players, and to guarantee that that the interaction among the players will be performed in the desired way. In contrast, the roles are separated from the class offering them in the relationship-role as attribute patterns, and, thus, roles cannot access the classes offering the roles. They only share the relationship namespace.

So the two patterns offer a tradeoff between the coupling of the role together with the class offering it (`Student` of a `Course`, `Employee` of an `Organization`, *etc.*), and the coordination among the roles. The ideal situation should allow both aspects to be dealt with. These problems are the result of the complexities concerning encapsulation arising when relatioships are taken seriously, as noted by Noble and Grundy [5] and reported in Section II.

A solution is possible in powerJava by exploiting an often disregarded feature of Java. The idea is as follows, and it is illustrated and in Figure 9 as an UML diagram. First, as in the relationship-role object pattern, a class for creating relationship objects is created, containing the roles (e.g., `Student` and `BasicCourse` in `AttendBasicCourse`), see Figure 10 . The interaction between the roles is defined at this level since the powers of each role can access the state of the other roles and of the relationship.

Differently from the relationship-role pattern, these roles must be defined as abstract and so cannot be instantiated. Moreover, the methods containing the details about how these methods describing interaction work can be left unfinished and declared as abstract.

Second, the same roles can be defined according to the relationship-role as attribute pattern in the classes offering them (and, thus, they can be developed separately), see Fig-

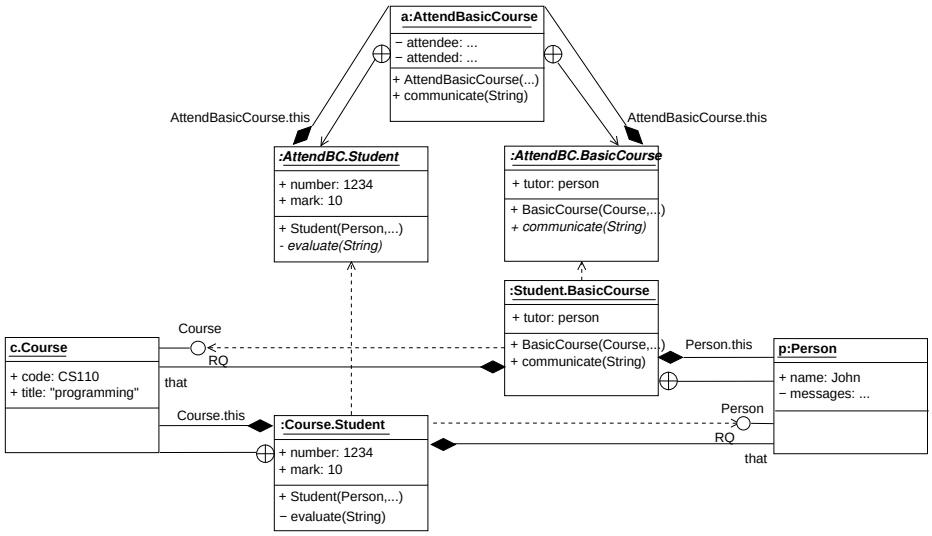

Fig. 9. The UML representation of the new relationship-role pattern

ure 11. However, these roles, rather than being implemented from scratch (e.g., `Student` and `BasicCourse`), they extend the abstract roles of the relationship object class, filling the gaps left by abstract methods in the abstract roles. The extension is necessary to customize the roles to their context. Methods which are declared as final in the abstract roles cannot be overwritten, since they represent the interaction among roles in the scope of the relationship. Further methods can be declared, but they are not visible from outside since both the abstract role and the concrete one have the signature of the role declaration.

Note that the abstract roles are not extended by the classes participating in the relationship (e.g., `Course` and `Person`), but by roles offered by (i.e., implemented into) these classes. Otherwise, the classes participating in the relationship could not extend further classes, thus limiting the code reuse possibilities.

The advantage of these solutions is that roles can share both the namespace of the relationship object class and the one of the class offering the roles, as we required above. This is possible since extending a role implementation is the same as extending an inner class in Java: roles are compiled into inner classes. When a class extends an inner class in Java, it maintains the property that the methods defined in the inner class it is extending continue to have access to the outer class instance containing the inner class. If the inner class is extended by another inner class, the resulting inner class belongs to the namespaces of both outer classes. Moreover, the inner class instance has a reference to both outer class instances so to be able to access their states. The possible ambiguities of identifiers accessible in the two outer classes and in the superclass are resolved by using the name of the outer class as a prefix of the identifier (e.g., `Course.this.registry`).

This feature of Java, albeit esoteric, has a precise semantics, as discussed by [19]. We exploit this mechanism for extending roles in relationship object classes, thus giving a new, more usable, meaning to this construct and hiding the complexities to the programmer.

Basing on this idea we propose here an extension of powerJava, which allows to define abstract roles inside relationship object classes, and to let standard roles extend them. The resulting roles will belong both to the namespace of the class offering them and to the relationship object class. Moreover, the resulting roles will inherit the methods of the abstract roles.

Note that the abstract roles cannot be instantiated, so that the are used only to implement both the methods which define the interaction among the roles, and the methods which are requested to be contextualized. The former will be final methods which are inherited, but which cannot be overwritten in the extending role: they will access the state and methods of the outer class and of the sibling roles. The latter will be abstract protected methods, which are used in the final ones, and which must be implemented in the extending class to tailor the interaction between the abstract role and the class offering the role. If these methods are declared as protected they are not visible outside the package. These methods have access to the class offering the extending roles.

Besides adding the property `abstract` to roles, three other additions are necessary in powerJava.

First of all, the methods of the abstract role can make reference to the outer class of the extending role. This is realized by means of a reserved variable `outer`, which is of type `Object` since it is not possible to know in advance which classes will offer the extended role. This variable is visible only inside abstract roles.

Second, to create a role instance it is necessary to have at disposal also the relationship object offering the abstract roles, and the two roles must be created at the same time. For

```
role Student playedby Person complements BasicCourse {
  int giveExam(String work); }

role BasicCourse playedby Course complements Student {
  void communicate(String text); }

class AttendBasicCourse{
  Student attendee;
  BasicCourse attended;

  abstract class Student {
    int mark;
    int number;
    //method modelling interaction
    final int giveExam(String work){
      return mark = evaluate(work);}
    //method to be implemented
    abstract protected int evaluate(String work);
  }

  abstract class BasicCourse {
    String program;
    Person tutor;
    //method to be implemented
    abstract void communicate(String text);
  }

  AttendBasicCourse(Person p, Course c, String pr, Person t){
    attendee = c.new Student(p,this);
    attended = p.new BasicCourse(c,this,t);
  }
}
```

Fig. 10. The new relationship-role pattern

example:

```
AttendBasicCourse(Person p, Course c){
  ...
  c.new Student(p,this);
  p.new BasicCourse(c,this);
}
```

Where `Student` and `BasicCourse` are the name of the concrete roles implemented in `p` and `c` and it is the same as the abstract roles defined in the relation.

The types of the arguments `Person` and `Course` are the requirements of the roles `Student` and `BasicCourse`.

Moreover, the first and the second argument of the constructor are added by default: the first one represents the player of the role, while the second one, present only in roles extending abstract roles, is the reference to the relationship object. This is necessary since the inner class instance represented by the role has two links to the two outer classes it belongs to. This reference is used to invoke the constructor of the abstract role, as required by Java inner classes, for example, the constructor of the role `Course.Student` is the following one.

```
Student(Person p, AttendBasicCourse a){
  a.super();
  ... }
```

However, these complexities are hidden by powerJava which adds the necessary parameters and code during precompilation.

The entities related by the relationship must preexist to it:

```
Person p = new Person();
Course c = new Course();
AttendBasicCource r =
  new AttendBasicCourse(p,c);
((c.Student)p).giveExam(w);
```

Note that the role cast `((r.Student)p)` is equivalent to `((c.Student)p)`.

Third, in the extension of powerJava abstract roles only come in pairs (e.g., `Student` and `BasicCourse`). Thus, the definition of a role must be extended to specify not only that the possible players comply with the requirements, but also which role must be offered in turn to play a role. E.g., a class `Person` to play the role `Student` has to offer in turn the role `BasicCourse`. For this reason, the role definition is extended with the keyword `complements` specifying the other role of the relation. E.g.,

```
role Student playedby Person
  complements BasicCourse {
    int giveExam(String work); }
```

Thus, a class can play a role not only if it implements the requirements in the role definition, but also if it offers the role specified as complementary.

Finally, we add an additional constraint to powerJava: if a role implementation extends another role, it must have the

```
class Course {
  String code;
  String title;
  private HashSet<Student> attendees;
  private HashTable registry = new HashTable();

  class Student extends AttendBasicCourse.Student {
    Student(){
      Course.this.attendee = this;
    }
    //abstract method implementation
    protected int evaluate(String work){
      mark = ...
      Course.this.registry.set(that.hashCode(), mark);
      return mark;
    }
  }
}

class Person {
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended; //courses followed as BasicCourse

  class BasicCourse extends AttendBasicCourse.BasicCourse{
    BasicCourse(Person t){
      tutor=t;
      Person.this.attended=this;
    }
    //abstract method implementation
    void communicate (String text) {
      Person.this.messages.add(text);
    }
  }
}
```

Fig. 11. The new relationship-role pattern

same name. Thus, the abstract and concrete role have the same requirements. Moreover, it is possible to extend only abstract roles, while general inheritance among roles is not discussed here.

In Figures 10 and 11, we report the example used in the previous sections using the new pattern. Note in particular, that the class `Person` does not have anymore a `getMessage` method, like in the example of Figure 7, since the role `BasicCourse` of `Person` has access directly to the private queue of messages of a person. Moreover, the `registry` of exams in a `Course` can be updated when giving an exam, as in the relationship as attribute solution, since the role `Student` has access to the class offering it. Finally, the method `evaluate` is defined inside the role implementation `Student` of the class `Course`, so that it can be tailored to different kind of courses.

This example, however, does not show how the interaction among roles can be separated from the classes of the players and gathered inside the relationship object class. For this reason we add also another example.

Consider a relationship like a negotiation protocol `CNP` (Contract Net Protocol). It relates two objects: first, an object of type `Manager` which plays the role of `Initiator` and, as `Initiator`, makes calls for proposals `cfp`; second, an object of type `Worker`, who is able to execute a task, can play the role of `Participant` and, as such, to make `proposal`s in return to the `cfp`. Note that as in the relationship-role as attributes pattern, the `Initiator` role is offered by `Worker` to allow the `Manager` to call the method `cfp` to interact with the `Worker`. Viceversa, the role `Participant` is offered by the `Manager` to the `Worker` to respond with a `proposal` to the `Manager`. However, differently than in the relationship-role as attributes pattern all the interaction among the roles happens inside the abstract roles defined in the class `CNP`. In this way, objects entering a negotiation are guaranteed that the role offered by the other participant is coherent with the one offered by their own. The only function of the role `Manager.Participant` and `Worker.Initiator` is to tailor the behavior of the abstract roles to the classes offering their extensions.

VII. Conclusion

In this paper we discuss why roles need to be introduced when relationships are modelled in OO programs: it is possible to distinguish between the natural type of objects populating the program and the state and behaviors they acquire when

```
role Initiator playedby InitiatorReq complements Participant {
   void cfp(Task task);
   void rejectProposal(Proposal proposal);
   void acceptProposal(Proposal proposal);
   }

role Participant playedby ParticipantReq complements Initiator {
  void propose(Proposal proposal);
  void refuse();
  void inform(String result);
  void failure();
  }

public class CNP {
  final static int STATE_1 = 1;
  final static int STATE_2 = 2; //...
  int state = STATE_1;

  abstract definerole Initiator {
    public final void cfp(Task task) throws IllegalPerformativeException {
      if (state != STATE_1) throw new IllegalPerformativeException ();
      state = STATE_2;
      if (evaluateTask(task))
        ((that.Participant)outer).propose(getProposal(task));
      else
        ((that.Participant)outer).refuse();
    }
    public final void rejectProposal(Proposal proposal) throws ... {
      if (state != STATE_3) throw new IllegalPerformativeException();
      state = STATE_4;
    }
    public final void acceptProposal(Proposal proposal) throws ... {
      if (state != STATE_3) throw new IllegalPerformativeException();
      if (performTask(proposal, task))
        ((that.Participant) outer).inform(performTask(proposal,task));
      else ((that.Participant)) outer).failure(error);
      state = STATE_5;
    }//methods to be implemented
    abstract protected boolean evaluateTask(Task task);
    abstract protected String performTask(Proposal proposal, Task task);
    }

  abstract definerole Participant {
    public final void propose(Proposal proposal) throws ... {
      if (state != STATE_2) throw new IllegalPerformativeException();
      state = STATE_3;
      if (evaluateProposal(proposal))
        ((that.Initiator)outer).acceptProposal(proposal);
      else
        ((that.Initiator)outer).rejectProposal(proposal);
      }
    public final void inform(String s) throws ... {
      if (state != STATE_2) throw new IllegalPerformativeException();
      state = STATE_3;
    }
    public final void refuse() throws ... {
      if (state != STATE_2) throw new IllegalPerformativeException();
      state = STATE_6;
    }
    public final void failure() throws ... {
      if (state != STATE_2) throw new IllegalPerformativeException();
      state = STATE_7;
    }
    abstract protected boolean evaluateProposal(Proposal proposal);
  }
}
```

Fig. 12. The CNP example

```
class Manager implements InitiatorReq{

  definerole Participant extends CNP.Participant {
    protected boolean evaluateTask(Task task){...}
  }
}

class Worker implements ParticipantReq {

  definerole Initiator extends CNP.Initiator{
    protected String performTask(Proposal proposal, Task task){...}
    protected boolean evaluateProposal(Proposal proposal){...}
   }
}

public static void main (String[] args){
  Worker w = new Worker();
  Manager m = new Manager();
  CNP c = new CNP(m,w);
  try{((w.Initiator)m).cfp(...);}
  catch (IllegalPerformativeException e){}; }
}
```

Fig. 13. The CNP example

they participate in a relationship. The state and behaviors which are dynamically acquired are modelled by roles.

Using the language powerJava, a role endowed version of Java, we show how to introduce roles in the two major patterns for modelling relationships: the relationship as attribute pattern and the relationship object pattern. We discuss the pros and cons of both patterns when roles are introduced. In particular, we show that the relationship as attribute pattern extended with roles enables to model the extension of behavior of the objects entering a relationship, without the introduction of a further class modelling the relationship.

The two resulting patterns differ also for the fact that the former emphasise the coupling of the role with the class offering it (e.g., `Student` and `Course`), while the latter emphasise the coupling of the roles with the relationship class and with each other.

Finally we propose a new pattern where both couplings can be considered at the same time: first abstract roles are defined in the relationship object class, which specify the interaction, and then the roles are defined in the classes offering them. This pattern solves the encapsulation problems raised when relationship are introduced in OO.

Future work includes studying how to introduce roles for relationship patterns developed for aspect programming, like the one proposed by Pearce and Noble [12].

REFERENCES

[1] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language." in *Procs. of OOPSLA*, 1987, pp. 466–481.

[2] J. Noble, "Basic relationship patterns," in *Pattern Languages of Program Design 4*. Addison-Wesley, 2000.

[3] G. Bierman and A. Wren, "First-class relationships in an object-oriented language." in *Procs. of ECOOP*, 2005, pp. 262–286.

[4] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini, "An object data model with roles," in *Procs. of Very Large DataBases (VLDB'93)*, 1993, pp. 39–51.

[5] J. Noble and J. Grundy, "Explicit relationships in object-oriented development," in *Procs. of TOOLS 18*, 1995.

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

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

[8] M. Baldoni, G. Boella, and L. van der Torre, "Roles as a coordination construct: Introducing powerJava," *Electronic Notes in Theoretical Computer Science*, vol. 150, no. 1, pp. 9–29, 2006.

[9] ——, "Modelling the interaction between objects: Roles as affordances," in *Procs. of Knowledge Science, Engineering and Management, KSEM'06*, ser. LNCS, vol. 4092. Berlin: Springer, 2006, pp. 42–54.

[10] ——, "Interaction among objects via roles: sessions and affordances in powerjava," in *Procs. of PPPJ '06*. New York (NY): ACM, 2006, pp. 188–193.

[11] ——, "Interaction between objects in powerJava," *Journal of Object Technology*, vol. 6, no. 2, pp. 7–12, 2007.

[12] D. Pearce and J. Noble, "Relationship aspects." in *Procs. of AOSD*, 2006, pp. 75–86.

[13] F. Steimann, "A radical revision of UML's role concept," in *Procs. of UML2000*, 2000, pp. 194–209.

[14] OMG, *OMG Unified Modeling Language Specification, Version 1.3*, 1999.

[15] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, 1999.

[16] B. Kristensen and K. Osterbye, "Roles: conceptual abstraction theory and practical language issues," *Theory and Practice of Object Systems*, vol. 2, no. 3, pp. 143–160, 1996.

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

[18] A. Whitehurst, "Association frameworks in simulation reuse," in *Procs. of OOS*, 1998.

[19] M. Smith and S. Drossopoulou, "Inner classes visit aliasing," in *ECOOP 2003 Workshop on Formal Techniques for Java-like Programming, 2003*, 2003.