The Interplay between Relationships, Roles and Objects

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
Computer Science and Communication University of Luxembourg
leendert@vandertorre.com

Abstract

In this paper we study the interconnection between relationships and roles. We start from the patterns used to introduce relationships in object oriented languages. We show how the role model proposed in powerJava can be used to define roles in an abstract way in objects representing relationships, to specify the interconnections between the roles. Abstract roles cannot be instantiated. To participate in a relationship, objects have to extend the abstract roles of the relationship. Only when roles are implemented in the objects offering them, they can be instantiated, thus allowing another object to play those roles.

1. 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]: he 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 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 ontologist 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 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.

We introduce roles in OO programming languages, in an extension of the Java programming language, called powerJava, described in [7, 8, 9, 10, 11]. 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 (e.g., a `Course` offers the role `Student` to a `Person` which wants to interact with it), i.e., the methods they can call and the state of interaction. 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. Roles in powerJava are essentially inner classes which are linked not only to an instance of the outer class, called institution, but also to an instance representing the player of the role. The player of the role, to invoke the methods of the roles it plays, it has to be casted to the role, by specifying both the role type and the institution it plays the role in (e.g., the university in which it is a student).

In [12] we add roles to the relationship as attribute pattern: the relationship is modelled as a pair of roles (e.g., attending a course is modelled by the role `Student` played by `Person` and `BasicCourse` played by `Course`) instead of a pair of links, like in the original pattern. In this way, the state of the relationships and the new behavior resulting from entering the relationship can be modelled by the fact that roles are adjunct instances with their state and behavior.

However, that solution fails to capture the coordination between the two roles, since in this pattern the roles are defined independently in each of the objects offering them (`Person` offering `BasicCourse` and `Course` offering `Student`) as we discuss in Section 4. This is essentially an encapsulation problem, raised by the presence of a relationship.

In this paper, we provide a solution to this limitation by introducing abstract roles defined by relationships and extended by roles of objects offering them. When roles are defined in the relationships, the interconnection between the roles can be specified (e.g., the methods describing the protocol the roles use to communicate). When roles are extended in the objects offering them, they can be customized to the context. Roles defined in the relationships are abstract and thus they cannot be instantiated. Roles can be instantiated only when they are extended in the objects which will participate to the relationship.

2. 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.

Pearce and Noble [13] 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."

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 [16], 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 [17] 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 in order to be instantiated. However, given that in a language like Java multiple inheritance is not allowed, this solution is not viable, and roles can be identified with interfaces only.

In this paper, we overcome the problem of the lack of multiple inheritance, by allowing objects participating to the relationship to offer roles which inherit from abstract roles related to the relationship, rather than imposing that objects extend the roles themselves. This is made possible by powerJava.

```
class Printer {
  private int totalPrintedPages = 0;
  private void print(Job job, Login login) {
    ... // performs printing
    totalPrintedPages += job.getNumberPages();
  }
  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 1. The `Printer` class and its affordances

3. powerJava: roles in Java

Baldoni *et al.* [7, 8, 9, 10, 11] introduce roles as affordances [18] in powerJava, an extension of the object oriented programming language Java. The notion of affordance comes from cognitive science and it refers to subjective properties and behavior of objects which emerge only during the interaction with a specific kind of entities, depending on their capabilities of interaction. 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 player objects:

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

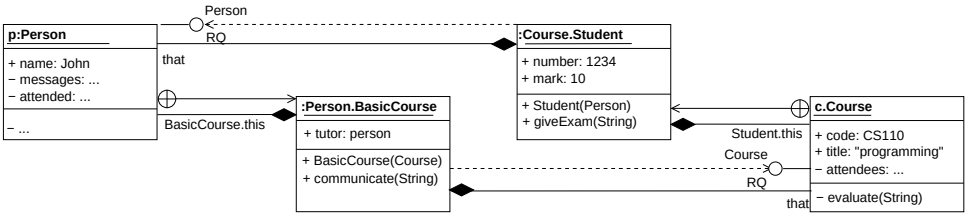

Figure 2. The UML representation of the relationship as attribute with roles pattern example

Instead, affordances, as subjective possibilities of interaction, 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 3). 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 [8] 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.

When an object uses the methods offered by a role, it should be able to invoke them without any explicit reference to the instance of the role. In this way the association between the object instance and the role instance is transparent to the programmer. The object should only specify in which role it is invoking the method. For example, if a person is a `User` and it has to print something, we want it to be able to invoke the method `print` on the person as a `User` without referring to the role instance. Note that this does not exclude the possibility of assigning the reference to a role instance to a variable ad then use it for invoking the role methods (see the variable `user` in the code below). Roles are always roles in an institution. Hence, an object can play at the same moment the same role more than once, albeit in different institutions. Instead, we do not consider the case of an object playing the same role more than once in the same institution. An object can play several roles in the same institution. In order to specify the role under which an object is referred, we evocatively use the same terminology used for casting by Java: we say that there is a casting from the object to the role. However, to refer to an object in a certain role implementation, both the object and the institution where it plays the role must be specified, thus reflecting the foundation property. We call this methodology *role casting*. Role casting is a means for stating that an object will act according to the powers that allow it to interact in a given institution. In the following the two `Users` invoke method `print` on `hp8100`. Notice that the page counter is maintained in the role state and persists through different calls to methods performed by a same player towards the same institution as long as it plays the role.

```
((hp8100.User) chris).print(job1);
((hp8100.SuperUser) sergio).print(job2);
System.out.println("Chris has printed " +
  ((hp8100.User) chris).getPrintedPages() + " pages");
System.out.println("hp8100 has printed " +
  ((hp8100.User) sergio).getTotalPrintedPages() +
    " pages");

User user = ((hp8100.User) chris);
user.print(job3);
System.out.println("Chris has printed " +
  ((hp8100.User) chris).getPrintedPages() + " pages");
```

Supposing that `job1` consists of ten pages, `job2` of twenty pages and `job3` of fifteen, two first output line will print ten, the second thirty (the sum of the lengths of `job1` and `job2`), the third twentyfive (the sum of `job1` and `job3`). By maintaining a state, a role can be seen as realizing a *session-aware interaction*, in a way that is analogous to what done by cookies or Java sessions for JSP and Servlet. So in our example, it is possible to visualize the number of currently printed pages by the user `chris`.

Since an object can play multiple roles, the same method will have a different behavior, depending on the role which the object is playing when it is invoked. However, there will be no conflict among roles, since only the powers of one role at a time can be exercised. To play a role it is sufficient to specify which is the role of a given object we are referring to. In the next example `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 in the role instance is incremented.

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

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*.

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.

```
role Student playedby Person
   { int giveExam(String work); }
role BasicCourse playedby Course
   { void communicate(String text); }

class Person{
  String name;
  private Queue messages;
  //BasicCourses followed
  private HashSet<BasicCourse> attended;
  definerole BasicCourse {
    Person tutor;
    // method access the state of outer class
    void communicate (String text)
      { Person.messages.add(text); }
    BasicCourse(Person t){
      tutor=t;
      Person.attended.add(this); }}//add link
}
class Course {
  String code;
  String title;
  //students of the course
  private HashSet<Student> attendees;
  private int evaluate(String x){...}
  definerole Student {
    int number;
    int mark;
    int giveExam(String work)
     { return mark = Course.evaluate(work); }
    Student () //add link
      { Course.attendees.add(this); }}}
```

Figure 3. Relationship as attribute pattern with roles in powerJava

4. Relationship as attribute with roles pattern

We first describe how the relationship as attribute pattern can be extended with roles, and second the relationship object pattern with roles. Then, starting from the limitation of these new patterns, in Section 6 we define a new solution introducing abstract roles in relationships. As an example we will use the situation where a `Person` can be a `Student` and follow a `Course` as a `BasicCourse` in his curriculum.

In [12], the relationship as attribute pattern is extended with roles by reducing the relationship not only to two symmetric attributes `attended` and `attendees` but also to a pair of roles. E.g., a `Person` plays the role of `Student` with respect to the `Course` and the `Course` plays the role of `BasicCourse` with respect to the `Person` (see Figure 2 and 3, where the UML representation is illustrated[1]).

The role `Student` is associated with players of type `Person` in the role specification (`role`), 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 `Course` allows a `Person` to interact with itself, 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 or calls its private methods, thus violating the standard encapsulation.

[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.

```
role Student playedby Person
  { int giveExam(String work); }
role BasicCourse playedby Course
  { void communicate(String text); }

class Person{
 String name;
 Queue messages;
 void getMessage(String text)
   {messages.add(text)};
}
class Course {
  String code;
  String title;
}
class AttendBasicCourse{
  Student attendee;
  BasicCourse attended;
  static Hashset<AttendBasicCourse> all;
  definerole Student {
    int mark;
    int number;
    int giveExam(String work){
      mark=
        AttendBasicCourse.attended.evaluate(work);}
  }
  definerole BasicCourse {
    String program;
    Person tutor;
    private int evaluate(String work){...}
    void communicate(String t){
      //invoke the requirement of the player
      AttendBasicCourse.attendee.that.getMessage(t);}
    }
  AttendBasicCourse(Person p, Course c, String p,
    Person t){
    attendee = this.new Student(p);
    attended = this.new BasicCourse(c,p,t);
    AttendBasicCourse.all.add(this);
  }
  static void communicate(String text){
   for (AttendBasicCourse x: all)
     x.attended.communicate(text);}
}
```

Figure 4. Relationship object with roles pattern, part I

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 methods of roles, called *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. if `Course c`: `c.new Student(p)`.

When the player of a role invokes a method of a role, 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(...)`.

This pattern with roles allows to add state and behavior to a relationship between `Person` and `Course`, without adding a new class representing the relationship. The limitation of this pattern is that the two roles `Student` and `BasicCourse` are defined independently in the two classes `Person` and `Course`. Thus, there is no warranty that they are compatible with each other (e.g., they communicate using the same protocol, despite the fact that they offer the methods specified in the role specification). Moreover, we would like that all roles of a relationship can access the private state of each other (i.e., share the same namespace). However, this would be feasible only if the two roles `Student` and `BasicCourse` are defined by the same programmer in the same context. This is not possible since the two player classes `Person` and `Course` may be developed independently.

```
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 of Course gives the exam
    ((a.Student)p).giveExam(work);
    //c's message to  Student of  Course
    ((a.BasicCourse)c).communicate(text);}}
```

Figure 5. Relationship object with roles pattern, part II

5. Relationship object pattern

The alternative relationship object with roles pattern introduces an `AttendBasicCourse` class modelling the relationship between `Person` and `Course`. However, the `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 4, 5 and 6). Like in the previous solution the roles are modelled as inner 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`.

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

Advantages of the relationship as attribute with roles 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 as attribute with roles 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.

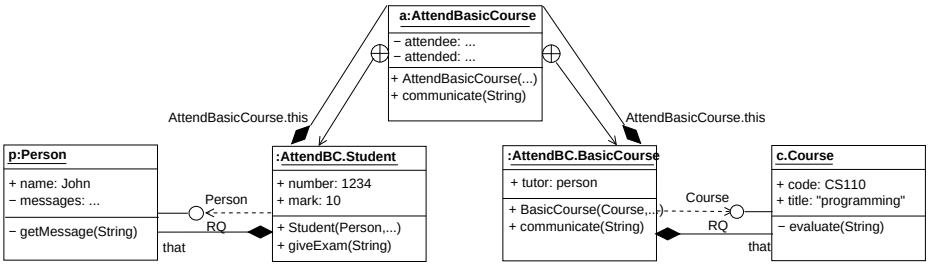

Figure 6. The UML representation of the relationship object with roles pattern example

- 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 with roles 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 [7].
- 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 object with roles 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, to play a role, an object is required to offer additional methods (see `getMessage` in Figure 4).

In summary, we would like:

- to define the interaction between the roles separately from the classes offering them to participate in the relationship, thus to guarantee that the interaction between the objects eventually playing the roles is performed in the desired way;
- that the roles of a relationship have access to the private state of each other to facilitate their programming;
- that the roles have also access to the private states of the objects offering them (like in powerJava) to customize them to the context.

```
role Student playedby Person
  { int giveExam(String work); }
role BasicCourse playedby Course
  { void communicate(String text); }

class AttendBasicCourse {
  Student attendee;
  BasicCourse attended;
  abstract definerole Student {
    int mark;
    int number;
    //method modelling interaction
    final int giveExam(String work){
      return mark = evaluate(work);}
    //method to be implemented which is not public
    abstract protected int evaluate(String work);
  }
  abstract definerole BasicCourse {
    String program;
    Person tutor;
    //method to be implemented which is public
    abstract void communicate(String text);
  }
  AttendBasicCourse(String pr, Person t){
    attendee = c.new Student(p,this);
    attended = p.new BasicCourse(c,this,t);
  }
}
```

Figure 7. Abstract roles

These requirements mirror the complexities concerning encapsulation, which arise when relationships are taken seriously, as noticed by Noble and Grundy [5].

6. Abstract roles and relationships

A solution to the encapsulation problem is possible in powerJava by exploiting an often disregarded feature of Java. Inner classes share the namespace of the outer classes containing them. When a class extends an inner class in Java, it maintains the property that the methods defined in the inner class which it is extending continue to have access to the private state of 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, an instance of such an inner class 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.registry`).

This feature of Java, albeit esoteric, has a precise semantics, as discussed by [19].

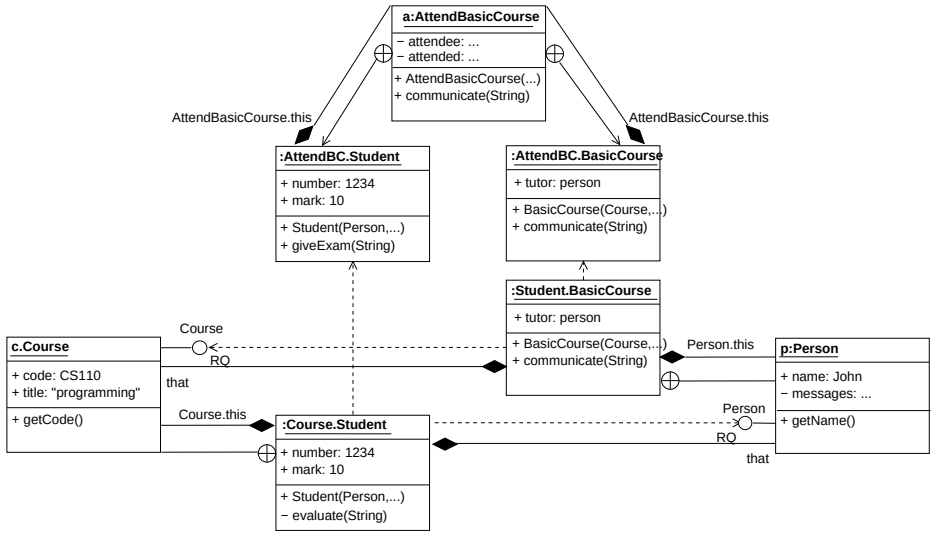

Figure 9. The UML representation of the new relationship pattern

```
class Course {
  String code, title;
  private HashSet<Student> attendees;
  class Student extends AttendBasicCourse.Student {
    Student() {
      Course.this.attendee = this;
    }
    //abstract method implementation
    protected int evaluate(String work)
      { /*Course specific
           implementation of the method */  }  } }

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);} } }
```

Figure 8. Abstract roles extended

The new solution we propose allows to introduce a new class representing the relationship as in the relationship object with roles pattern, and to define the roles inside it. The idea is illustrated and in Figure 9 as an UML diagram.

First, as in the relationship object with roles pattern, a class for creating relationship objects is created (e.g., `AttendBasicCourse`): it will contain the implementation of the roles involved in the relationship (e.g., `Student` and `BasicCourse` in `AttendBasicCourse`), see Figure 7. 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.

These roles must be defined as abstract and so they cannot be instantiated. Moreover, the methods containing the details about the customization of the role can be left unfinished (i.e., declared as abstract) if they need to be completed depending on the classes offering the roles which extend the abstract roles.

Second, the same roles in the relationship can be implemented in the classes participating in the relationship (and, thus, they can be extended separately), accordingly to the relationship as attribute pattern, see Figure 8 (`Person` offering `BasicCourse` and `Course` offering `Student`). However, these roles (e.g., `Student` and `BasicCourse`), rather than being implemented from scratch, extend the abstract roles of the relationship object class (e.g., `AttendBasicCourse`), filling the gaps left by abstract methods in the abstract roles (both public and protected methods). The extension is necessary to customize the roles to their new 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 (e.g., `Student` and `BasicCourse`). Otherwise, the classes participating in the relationship could not extend further classes, since Java does not allow multiple inheritance, thus limiting the code reuse possibilities.

The advantage of these solution 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 by the powerJava precompiler.

Basing on this idea we propose here a limited 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 eventual 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, we add an additional constraint to powerJava: if a role implementation extends an abstract role, it must have the 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.

Second, 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.

Third, 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 example, the constructor of a relationship:

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

Where `Student` and `BasicCourse` are the class names of the concrete roles implemented in `p` and `c` and they are 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` which will be used to type the `that` parameter referring to the player of the role.

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 class instances 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);
((p.BasicCourse)c).communicate(text);
```

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

7. Conclusion

In this paper we discuss how abstract roles can be introduced when relationships are modelled in OO programs: first abstract roles are defined in the relationship object class, which specify the interaction, and then the abstract roles are extended in the classes offering them. This pattern solves the encapsulation problems raised when relationships are introduced in OO.

We introduce abstract roles using the language powerJava, a role endowed version of Java (`http://www.powerjava.org`) [7, 8, 9, 10, 11, 12].

References

[1] Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: Procs. of OOPSLA. (1987) 466–481

[2] Noble, J.: Basic relationship patterns. In: Pattern Languages of Program Design 4. Addison-Wesley (2000)

[3] Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Procs. of ECOOP. (2005) 262–286

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

[5] Noble, J., Grundy, J.: Explicit relationships in object-oriented development. In: Procs. of TOOLS 18. (1995)

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

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

[8] Baldoni, M., Boella, G., van der Torre, L.: powerJava: ontologically founded roles in object oriented programming language. In: Procs. of OOPS Track of ACM SAC'06, ACM (2006) 1414–1418

[9] Baldoni, M., Boella, G., van der Torre, L.W.N.: Modelling the interaction between objects: Roles as affordances. In: Procs. of Knowledge Science, Engineering and Management, KSEM'06. Volume 4092 of LNCS., Springer (2006) 42–54

[10] Baldoni, M., Boella, G., van der Torre, L.: Interaction among objects via roles: sessions and affordances in powerjava. In: Procs. of PPPJ '06, New York (NY), ACM (2006) 188–193

[11] Baldoni, M., Boella, G., van der Torre, L.: Interaction between Objects in powerJava. Journal of Object Technology **6** (2007) 7–12

[12] Baldoni, M., Boella, G., van der Torre, L.: Relationships meet their roles in object oriented programming. In: Procs. of the 2nd International Symposium on Fundamentals of Software Engineering 2007 Theory and Practice (FSEN '07). (2007)

[13] Pearce, D., Noble, J.: Relationship aspects. In: Procs. of AOSD. (2006) 75–86

[14] OMG: OMG Unified Modeling Language Specification, Version 1.3. (1999)

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

[16] Steimann, F.: A radical revision of UML's role concept. In: Procs. of UML2000. (2000) 194–209

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

[18] Gibson, J.: The Ecological Approach to Visual Perception, Lawrence Erlabum Associates. New Jersey. (1979)

[19] Smith, M., Drossopoulou, S.: Inner classes visit aliasing. In: ECOOP 2003 Workshop on Formal Techniques for Java-like Programming. (2003)