

# Relationships Meet Their Roles in Object Oriented Programming

Matteo Baldoni<sup>1</sup>, Guido Boella<sup>1</sup>, and Leendert van der Torre<sup>2</sup>

<sup>1</sup> Dipartimento di Informatica - Università di Torino - Italy  
baldoni@di.unito.it guido@di.unito.it

<sup>2</sup> Computer Science and Communications (CSC) - University of Luxembourg  
leendert@vandertorre.com

**Abstract.** In this paper we study how roles can be added to patterns modelling relationships in Object Oriented programming. 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 the natural types of the participants in the relationships from the roles the participants acquire in the relationships. We show how the role model of the language powerJava can be used to endow the relationship as attribute pattern with 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]. Rumbaugh [1] claims that relationships are complementary to, and as important as, objects themselves. For example, a student can be related to a university by an enrollment relationship, he can attend a course, and have a tutor. Thus, relationships should not only be present in modelling languages, like ER or UML, but they should also be available in programming languages, either as primitives, or, at least, represented by means of suitable patterns.

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

- The relationship as attribute pattern: the relationship is modelled by means of an attribute of the objects which participate in the relationship. E.g., the `Attend` relationship between a `Student` and a `Course` can be modelled by means an attribute `attended` of the `Student` and of an attribute `attende` of the `Course`.
- The relationship object pattern: the relationship is modelled as a third object linked to the participants `Student` and `Course`. 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., [3].

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

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

**Fig. 1.** The relationship as attribute pattern

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], courses are basic courses and advanced courses [4], customers buy from sellers [5], employees are employed by employers, underwriters interact with reinsurers [2], *etc.* From the ontological point of view these concepts are not natural kinds like person or organization: rather, they all are roles involved in a relationship [6].

Roles have different properties than natural kinds, and, thus, are difficult to model with classes: they are dynamically acquired, they depend on other entities - the relationship they belong to and their players - and they add properties and behaviors to the objects playing roles. Moreover, roles can be played by objects of different classes. In particular, 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 exist only as far as he is a student of that course.

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 question: How to introduce roles in relationships? And as subquestion: Which are the advantages given by roles in the relationship as attribute pattern?

In this paper as methodology we use our model of roles in OO programming languages which has been added to an extension of the Java programming language, called powerJava, described in [8,9].

The language powerJava introduces roles as a way to structure the interaction of objects (callee) with other objects calling their methods (caller). Roles express the possibilities of interaction offered by a callee to another one, i.e., the methods it can call. First, these possibilities change according to the class of the caller of the methods. Second, a role maintains the state of the interaction with a certain 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 added to model relationships, where the behavior of an object changes when it enters a relationship until it subsequently abandons it.

In Section 2 we discuss how relationships are introduced in OO programming. In Section 3 we summarize our model of roles in powerJava and in Section 4 we use it to introduce roles in the relationship as attribute pattern.

## 2 Introducing Relations in OO

We will describe in this section the relationship as attribute pattern with reference to a university domain. Consider a student who can attend different kinds 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 his mark is reported on a registry, 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 a given course.

The relationship as attribute pattern is described in Figure 1: the relationship between a student and a 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 `BasicCourses`. Symmetrically, the `Student` appears in the attribute `attendees` of the type set of `Students` in the class `BasicCourse`.

This solution, however, does not allow to add a state and behavior to the pairs of elements related by the relationship. For example, it is not possible to specify a different tutor for each `Student` of the `BasicCourse`.

This is possible in the alternative pattern, the relationship object, where the participants in the relationship are linked to an object representing a relationship instance. This alternative solution can be modelled in UML, which specifies information proper of an association via an association class, which can be endowed with properties and behaviors. 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.

But the relationship object solution shares with the relationship as attribute a limitation. We would like to model the scenario introducing natural types like `Person` rather than the `Student` class only. The reason for such modelling choice is that a `Person` is not always a `Student`, but only as long as he attends courses. Moreover, he can give exams or receive communications concerning the course, only if he is registered and, thus, related by the relationship with a `Course` which he follows as a `BasicCourse`. He has different marks in different exams, and even different students can have different tutors for the same course. Analogously a `Course` has a tutor only if it plays the role of `BasicCourse`. Note that `Person` instances can play also other roles while they are `Student`, like, e.g., employee.

Moreover, even if the relationship object pattern allows to add new properties and behaviors, 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 and behaviors belong to the `Student` and which ones to the `Course`. All properties and behaviors are associated to the instance of the class representing the relationship.

We leave modelling this pattern for future work, even if its realization in powerJava is straightforward.

```

class Printer {
    private int printedTotal;

    definerole User {
        private int printed;
        public void print(){ ...
            printed = printed + pages;
            printedTotal = printedTotal
                + printed;
            Printer.print(that.getName());
        }
    }
}

role User playedby UserReq
{ void print();
  int getPrinted(); }
interface UserReq
{ String getName();
  String getLogin(); }
jack = new AuthPerson();
laser1 = new Printer();
laser1.new User(jack);
((laser1.User)jack).print();

```

Fig. 2. A role User inside a Printer

### 3 Roles in Powerjava

Baldoni *et al.* [8,9] introduce roles in powerJava, an extension of the object oriented programming language Java. Java is extended with:

1. A construct specifying the role with its name, the methods required to play the role, and the operations it offers to its players.
2. The implementation of a role, inside another object and according to its definition.
3. How an object can play a role and invoke the operations offered by the role.

Figure 2 shows the use of roles in powerJava by means of the example of a printer which can be accessed via roles, e.g. *User*. First of all, a role is specified as a sort of interface (*role* - right column) by indicating via an interface which classes can play the role (*playedby*) and which are the operations acquired by playing the role (e.g., *print*). Second (left column), a role is implemented inside an object as a sort of inner class which realizes the role specification (*definerole*). The inner class implements all the methods required by the role specification as it were an interface.

In the bottom part of the right column of Figure 2, the use of powerJava is depicted. First, the candidate player *jack* of the role is created. It implements the requirements of the roles (the class *AuthPerson* implements *UserReq*). Before the player can play the role, however, an instance of the object hosting the role must be created first (a *Printer laser1*). Once the *Printer* is created, the player *jack* can become a *User* too. Note that the *User* is created inside the *Printer laser1* (*laser1.new User(jack)*) and that the player *jack* is an argument of the constructor of role *User* of type *UserReq*, which becomes the value of the special variable *that*, thus allowing to refer to the player from the role implementation.

The player *jack* to act as a *User* must be first classified as a *User* by means of a so-called *role casting* (*((laser1.User) jack)*). Note that *jack* is not classified as a generic *User* but as a *User* of *Printer laser1*. Once *jack* is casted to its *User* role, it can exercise its powers, in this example, printing (*print()*). Such method is called a power since, in contrast with usual methods, it can access the state of

```

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; //BasicCourse followed
  definerole BasicCourse {
    Person tutor;
    // the method communicate access the state of the outer class
    void communicate (String text) {Person.this.messages.add(text);}
    BasicCourse(Person t){
      tutor=t;
      Person.this.attended.add(this); }//add link
  }
}

class Course {
  String code;
  String title;
  private HashSet<Student> attendees; //students of the course
  private HashTable registry = new HashTable();
  private int evaluate(String x){...}
  definerole Student {
    int number;
    int mark;
    int giveExam(String work){ mark = Course.this.evaluate(work);
      registry.set(that.name.hashCode(), mark); ... }
    Student (){ Course.this.attendees.add(this); }}//add link
  }
}

```

**Fig. 3.** Relationship-role as attribute pattern in powerJava

other objects: the role namespace shares the one of the object including the role (called institution). In the example, the method `print()` can access the private state of the `Printer` and invoke `Printer.print()` or modify `printedTotal`.

## 4 The Relationship-Role as Attribute Pattern

In this section we describe how a new pattern for modelling relationships with roles can be defined, in analogy with the relationship as attribute pattern. We will use the example of Section 2 to present it.

First of all, using `powerJava` we can distinguish natural types like `Person` and `Course` from the respective roles `Student` and `BasicCourse`. `Person` and `Course` become, respectively, `Student` and `BasicCourse` when they enter the relationship. Roles are represented in `powerJava` by instances dynamically associated with the players of the roles, which include the state and behaviors acquired by the

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

**Fig. 4.** Using the relationship-role as attribute in powerJava

players of the roles in the relationship (see Figures 3 and 5 where the UML representation is illustrated<sup>1</sup>).

Second, in the relationship as attribute pattern, a relationship is reduced only to two symmetric attributes `attended` and `attendees`. In the new pattern, the relationship is modelled also by means of a pair of roles implemented in the two classes representing the natural types. Thus, the attribute `attendees` in `Course` of type `Student` in `Course` becomes `Course.Student`, and its values will be role instances which are played by instances of type `Person`. The role `Student` is associated with players of type `Person` in the role specification, which specifies that a `Student` can give an exam (`giveExam`). Analogously, the attribute `attended` of `Person` becomes of type `Person.BasicCourse` and its values are associated with players of type `Course` as in the role specification, which specifies that a `Course` can communicate with the attendee.

The role `Student` is implemented locally in the class `Course` by the class `Course.Student`, and, viceversa, the role `BasicCourse` is implemented locally in the class `Person` by the class `Person.BasicCourse`. Note that this is not contradictory, since roles describe the way an object offers interaction to another one: a `Student` represents what a `Course` offers a `Person` to interact with it, and, thus, the role is implemented inside the class `Course`. Moreover the methods associated with the role `Student`, i.e., giving exams, and implemented in `Course.Student`, modify the state of the class including the role (`Course`) or call its private methods, thus violating the standard encapsulation. Analogously, the `communicate` method of `Person.BasicCourse`, implementing the method signature specified in the role `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 that powers have 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 4) where the player `p` is passed as a parameter.

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

---

<sup>1</sup> The arrow starting from a crossed circle, in UML, represents the fact that the source class can be accessed by the arrow target class.

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 pattern has 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:

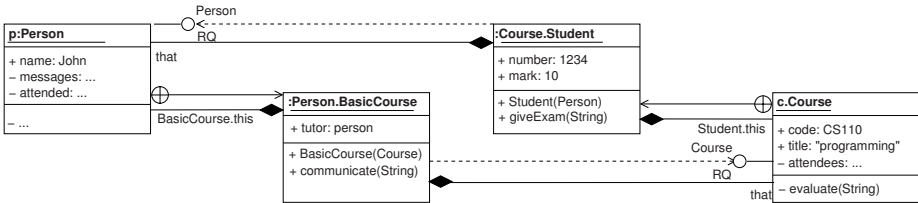
- It requires that the roles are already implemented offline inside the classes which participate in the relationship.
- It does not assure coherence of the pair of roles like student-course, buyer-seller, bidder-proponent, since they are defined separately in two different classes.
- The role cast to 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`).

In summary, we can define an informal program transformation, to add roles to the relationship as attribute pattern 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 kinds 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 role, or to an interface implemented by these natural types, and insert in the role specification the signature of the powers (e.g., `communicate`, `giveExam`).
4. Identify the two links to the participants in the relationships in the classes representing the participants (e.g., `attendees` of type `Student` in `Course`), now of natural types.
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`). Add to this role class the attributes and the implementation, according to the role specification, 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 players (of natural types), and, second, add these instances to the links modelling the relationship in the class of the players, e.g., `Person` (this can be done in the role constructors).

- 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.



**Fig. 5.** The UML representation of the relationship-role as attribute pattern example

## 5 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 roles they play, and objects acquire new states and behaviors when 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 the relationship as attribute pattern and we discuss the pros and cons of the pattern 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. Future work is introducing roles in the relationship as object pattern and designing new patterns where both patterns can be considered at the same time.

## References

- Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: *Procs. of OOPSLA*, pp. 466–481 (1987)
- Noble, J.: Basic relationship patterns. In: *Pattern Languages of Program Design 4*, Addison-Wesley, Reading (2000)
- Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Black, A.P. (ed.) *ECOOP 2005*. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)
- Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An object data model with roles. In: *VLDB 1993*. *Procs. of Very Large DataBases*, pp. 39–51 (1993)
- Noble, J., Grundy, J.: Explicit relationships in object-oriented development. In: Beilner, H., Bause, F. (eds.) *MMB 1995 and TOOLS 1995*. LNCS, vol. 977, Springer, Heidelberg (1995)

6. Masolo, C., Vieu, L., Bottazzi, E., Catenacci, C., Ferrario, R., Gangemi, A., Guarino, N.: Social roles and their descriptions. In: KR 2004. Procs. of Conference on the Principles of Knowledge Representation and Reasoning, pp. 267–277. AAAI Press, Stanford (2004)
7. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering* 35, 83–848 (2000)
8. Baldoni, M., Boella, G., van der Torre, L.: Interaction between Objects in powerJava. *Journal of Object Technology* 6, 7–12 (2007)
9. Baldoni, M., Boella, G., van der Torre, L.: Interaction among objects via roles: sessions and affordances in powerJava. In: PPPJ 2006. Procs. of Principles and Practice of Programming in Java, pp. 188–193. ACM, New York (2006)