

The Interplay between Relationships, Roles and Objects

Matteo Baldoni¹, Guido Boella², and Leendert van der Torre³

¹Dipartimento di Informatica - Università di Torino - Italy. email: baldoni@di.unito.it

²Dipartimento di Informatica - Università di Torino - Italy. email: guido@di.unito.it

³University of Luxembourg. e-mail: 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, and we show how the role model proposed in powerJava can be used to define roles. In particular, we focus on how to implement roles in an abstract way in objects representing relationships, and 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 `attende` 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 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 take 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 [7–10]. 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 course in which it is a student).

In [11] 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`). This is essentially an encapsulation problem, raised by the presence of a relationship.

In this paper, we provide a solution to this limitation first by extending the relationship object pattern with roles, and then 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 [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.”

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” [13]. 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 [14].

As noticed by Steimann [15], 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 [16] 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 printedTotal;
    private void print(){...}

    definerole User {
        private int printed;

        public void print(){ ...
            printed = printed + pages;
            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.new SuperUser(jack);
((laser1.User) jack).print();

```

Fig. 1. A role User inside a Printer.

3 Roles in powerJava

Baldoni *et al.* [10] introduce roles as affordances in powerJava, an extension of the object oriented programming language Java. Java is extended with:

1. A construct defining the role with its name, the requirements and the operations.
2. The implementation of a role, inside an object and according to its definition.
3. How an object can play a role and invoke the operations of the role.

Figure 1 shows the use of roles in powerJava. First of all, a role is specified as a sort of interface (*role* - right column) by indicating with an interface or class who can play the role (*playedby*) and which are the operations acquired by playing the role. 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 1 the use of powerJava is depicted. First, the candidate player *jack* of the role is created. It implements the requirements of the roles (*AuthPerson* implements *UserReq* and *SuperUserReq*). 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*. Moreover *jack* plays the role of *SuperUser*.

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 other objects: namespace shares the one of the object defining the role. In the example, the method *print()* can access the private state of the *Printer* and invoke *Printer.print()*.

4 Relationship as attribute with roles pattern

We first summarize how the relationship as attribute pattern is extended with roles in [11], and in the next section 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. Note that `BasicCourse` is not a subtype of `Course`, since a `Course` can be either a `BasicCourse` or an `BasicCourse` in different curricula.

In [11], 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 (see Figure 2). 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`.

The role `Student` is associated with players of type `Person` in the role specification (`role`), which specifies that a `Student` can take an exam (`takeExam`). 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. 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 `takeExam` 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).takeExam(...)`.

This pattern with roles allows state and behavior to be added 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

```

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

class Person{
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended; //BasicCourses followed
  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 takeExam(String work)
      { return mark = Course.evaluate(work); }
    Student () //add link
      { Course.attendees.add(this); }}}

```

Fig. 2. Relationship as attribute with roles pattern in powerJava

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.

The pattern has different pros and cons; the following list integrates Noble [2]'s discussions on them:

- 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 a state and operations to be introduced into 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 us 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.

```

class AttendBasicCourse{
    Student attendee;
    BasicCourse attended;
    static HashSet<AttendBasicCourse> all;
    definerole Student {
        int mark;
        int number;
        int takeExam(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);}
}

```

Fig. 3. Relationship object with roles pattern, part I

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.
- 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 us 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).

```

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

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

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

```

Fig. 4. 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 3, 4 and the UML diagram in Figure 5)¹. Like in the previous solution the roles are modelled as inner classes. In this example, the roles are implemented in the class `AttendBasicCourse`. Its 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

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.

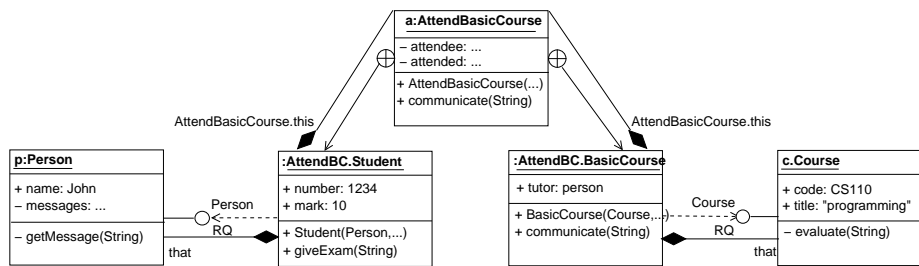


Fig. 5. The UML representation of the relationship object with roles pattern example

Student, a Person must be role casted to the role Student starting from an instance of the class AttendBasicCourse.

Advantages of the relationship role object with roles pattern:

- It allows a state and operations of the relationship to be introduced besides the state and operations added to the objects entering the relationship.
- It allows listing 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 3).

6 Abstract roles and relationships

From the above discussion, the following requirements emerge:

- to define the interaction between the roles separately from the classes offering them to participate in the relationship. This guarantees 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 also have access to the private states of the objects offering them (like in `powerJava`) to customize them to the context.

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

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 [17].

The new solution we propose allows the introduction of 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 in Figure 8 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 6. 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 7 (`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

```

role Student playedby Person
  { int takeExam(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 takeExam(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);
  }
}

```

Fig. 6. Abstract roles

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.

```

class Course {
    String code, title;
    private HashSet<Student> attendees;

    definerole 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

    definerole 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. 7. Abstract roles extended

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.

Based on this idea we propose here a limited extension of powerJava, which allows abstract roles to be defined 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. This is so that they 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

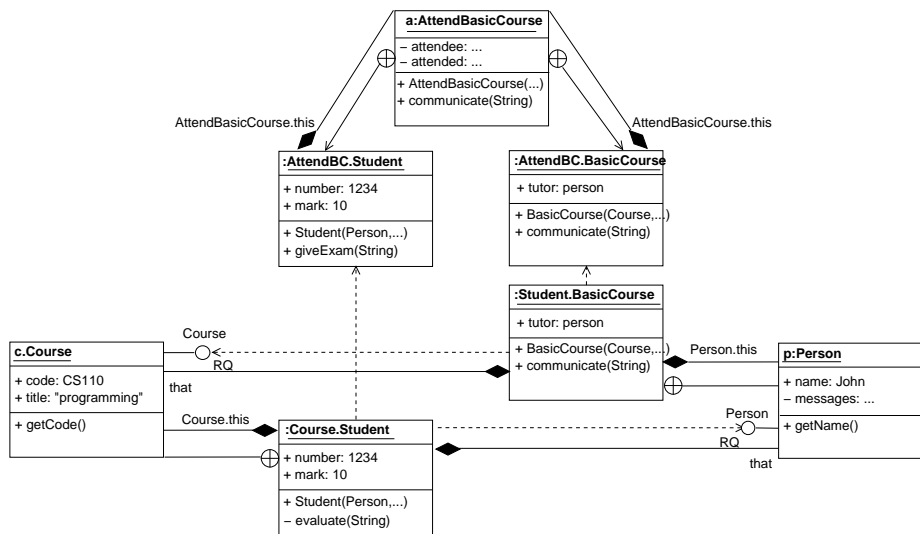


Fig. 8. The UML representation of the new relationship pattern

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 only possible to extend 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 also necessary to have at disposal 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();
AttendBasicCourse r = new AttendBasicCourse(p, c);
((c.Student)p).takeExam(w);
((p.BasicCourse)c).communicate(text);
```

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 ([7–11]).

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.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
9. 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
10. Baldoni, M., Boella, G., van der Torre, L.: Interaction between Objects in powerJava. *Journal of Object Technology* **6** (2007) 7–12
11. 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)
12. Pearce, D., Noble, J.: Relationship aspects. In: *Procs. of AOSD*. (2006) 75–86
13. OMG: *OMG Unified Modeling Language Specification, Version 1.3*. (1999)
14. Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*. Addison-Wesley (1999)
15. Steimann, F.: A radical revision of UML's role concept. In: *Procs. of UML2000*. (2000) 194–209
16. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering* **35** (2000) 83–848
17. Smith, M., Drossopoulou, S.: Inner classes visit aliasing. In: *ECOOP 2003 Workshop on Formal Techniques for Java-like Programming*. (2003)