Conceptual Modelling Patterns for Roles

Jordi Cabot[1,2] and Ruth Raventós[2]

[1]Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
Av. Tibidabo, 39-43, E08035 Barcelona
jcabot@uoc.edu
[2] Departament Llenguatges i Sistemas Informàtics, Universitat Politècnica de Catalunya
Campus Nord, Edif. Omega, Jordi Girona 1-3, E08034 Barcelona
raventos@lsi.upc.edu

Abstract. Roles are meant to capture dynamic and temporal aspects of real-world objects. The role concept has been used with many semantic meanings: *dynamic class*, *aspect*, *perspective*, *interface* or *mode*. This paper identifies common semantics of different role models found in the literature. Moreover, it presents a set of conceptual modelling patterns for the role concept that include both the static and dynamic aspects of roles. In particular, we propose the *Role as Entity Types* conceptual modelling pattern to deal with the full role semantics. A conceptual modelling pattern is aimed at representing a specific structure of knowledge that appears in different domains. The use of these patterns eases the definition of roles in conceptual schemas. In addition, we describe the design of schemas defined by using the patterns in order to implement them in any object-oriented language.

1 Introduction

An accurate and complete conceptual modelling is essential for a correct development of information systems. Reusable conceptual schemas facilitate this difficult and time-consuming activity. The use of patterns is essential to increase the reusability in all stages of software development.

A pattern identifies a problem and provides the specification of a generic solution to that problem. The definition of patterns in conceptual modelling may be regarded in two different ways: conceptual modelling patterns and analysis patterns.

In this paper, we distinguish between a conceptual modelling pattern that is aimed at representing a specific structure of knowledge encountered in different domains (for instance the *MemberOf* relationship) and an analysis pattern that specifies a generic and domain-dependent knowledge required to develop an application for specific users (for instance a pattern for electronic marketplaces). Authors do not always make this distinction. For example, to Fowler, in [13], patterns correspond to our conceptual modelling patterns while to Fernandez and Yuan, in [12], patterns correspond to our definition of analysis patterns. For a further discussion on analysis patterns see Teniente in [43].

The goal of this paper is to propose a set of conceptual modelling patterns to facilitate the representation of roles in conceptual schemas.

Although definitions of the role concept abound in the literature of conceptual modelling [3, 58, 13, 18, 35,37] a uniform and globally accepted definition is not given. To sum up, we could say that roles are meant to capture the dynamic and temporal aspects of real-world objects.

Roles appear very frequently in conceptual schemas. They are useful to model some dynamic situations from the real world that are not well represented just with the basic modelling language constructs, such as entity types that present different properties or behaviour depending on the context where they are used. For instance, the properties of a person when playing the role of student are different from those when playing the role of Employee. Moreover, when asking for his email address the answer depends on the role/s he is playing, it may be his personal email address, his student email address or his work email address.

Despite its importance, the possibilities that conceptual modelling languages offer to deal with roles are very limited (see, for example, what UML supports in [9] and [39]). In short, they consider roles just as the name of a participant in a relationship type.

We identify common and different semantics of role models found in the literature and characterize the patterns in terms of the features they cover. This allows the designer to choose the pattern best suitable for his needs. We also discuss the design and implementation of conceptual schemas that use these patterns to facilitate their implementation in object-oriented languages.

Of particular importance is our *Role as Entity Types* pattern, useful to represent roles when the full expressiveness of the role concept is needed. The pattern covers the most well-known role semantics. In contrast with previous approaches, one of its advantages is its simplicity, since roles and their evolution are represented with already existing constructs (entity types and constraints). Therefore, roles are easily integrated in conceptual schemas.

The rest of this paper is organized as follows: the next section presents a set of basic patterns. Section 3 proposes the *Roles as Entity Types* pattern. Section 4 comments related work and compares it with our proposals. Finally, conclusions and further work are presented.

2 Conceptual Modeling Patterns for Roles

The aim of this section is to define and compare a set of patterns to specify roles in conceptual schemas (CSs) by using just the standard constructs offered by conceptual modelling languages. Mosse in [24] and Fowler in [13] also propose a series of patterns for role representation. However, both of them discuss the patterns very informally and consider only a small subset of the role features we take into account.

One of the advantages of patterns explained in this section is that they are quite simple but, as a trade-off, their expressiveness is rather limited.

In order to describe the role patterns we adopt the template proposed by Geyer-Schulz and Hahsler in [16] to describe conceptual modelling patterns (called by the

authors *analysis patterns*). They adopt a uniform and consistent format, in contrast to Fowler in [13] who uses a very free format for pattern writing. Geyer-Schulz and Hahsler stress that adhering to a structure for writing patterns is essential since patterns are easier to teach, learn, compare with, write and use once the structure has been understood.

Their template preserves the typical context/problem/forces/solution structure of design patterns but adapted for the description of conceptual modelling patterns. The template includes the following sections: (1) Pattern Name. (2) Intent: what the pattern does and the problems it addresses. (3) Motivation: a scenario that illustrates the problem and how the pattern contributes to the solution in the specific scenario. (4) Forces and Context that should be resolved by the pattern. (5) Solution: description of all relevant structural and behavioural aspects of the pattern. (6) Consequences: how the pattern achieves its objectives and the existing trade-off. (7) Design and implementation: how the pattern can be realized in the design stage. (8) Known uses: examples of the pattern.

Obviously sections about intent, motivation and forces and context are common to all patterns for role representation. We first address these sections. Then, we define the solution that each pattern proposes, its consequences, its design and the known uses.

2.1 Intent

The intent is the representation of roles that entities play through their life span and the control of their evolution.

2.2 Motivation

The role concept appears frequently in many different domains of the real world, since we can find entity types in each domain that present some properties that evolve over time (see Papazoglou et al. in [32] and Jodlowski et al. in [20] for some example applications where roles are specially useful).

There is not a uniform and globally accepted definition of roles. The first commonly credited definition of roles in a data model goes back to the 70s when Bachman and Dayas proposed the *role data model* [3]. They defined a role as "a defined behaviour pattern that may be assumed by entities of different kinds". Since then, many other definitions and additional semantics have been proposed.

For instance, to Dahchour et al. in [8], the concept of role is "a generic relationship for conceptual modeling that relates a class of objects (e.g., people) and classes of roles (e.g., students, employees) for those objects. The relationship is meant to capture temporal aspects of real-world objects". To Papazoglou and Krämer in [31] a role "ascribes properties that possibly vary over time. The purpose of a role is to model different representation alternatives for the same object in terms of both structure and behavior".

To sum up, we could state that roles are useful to model the properties and behaviour of entities that evolve over time. The entity type *Person* is an illustrative

example. During his or her life, a person may play different roles, for example he or she may become a student, an employee, a project manager, and so forth. Besides this, a person may have different properties and behaviours depending on the role or roles he/she is playing at a certain time.

For instance, let us consider the following scenario, which will serve as a recurrent example in the following pages: let Maria be a person (with a name, phone number, birthdate, country, age, sex and full address), who starts a degree (Maria plays the role of student). After some years of study, she registers to a second university program (Maria plays twice the role of student) and starts to work in a company (Maria plays the role of employee). In that company she may become a project manager (now, Maria through her employee role, plays the role of project manager). If in the future, Maria became a department manager, now Maria through her employee role would play a new role, department manager.

For each role we are interested in recording a set of properties specific for that role. As employee we are interested in: her employee number, category, company phone number, working status and the expiration date of her contract. As a project manager we are interested in information about the project she manages (the project name, the start date and the tasks it involves). Moreover, roles share properties with the main entity type. For instance, when considering Maria as employee we also want to know her name, even though the name is not an explicit property of employee.

Figure 1 shows the different relationships that are involved in the scenario introduced above. Note that, in this situation, if we ask for the value of a property of Maria the answer is not trivial because it depends on the role or roles she is playing. For instance, if we ask for her telephone number, the answer may be her personal number (since Maria is a person) or her company phone number (since Maria is an employee).

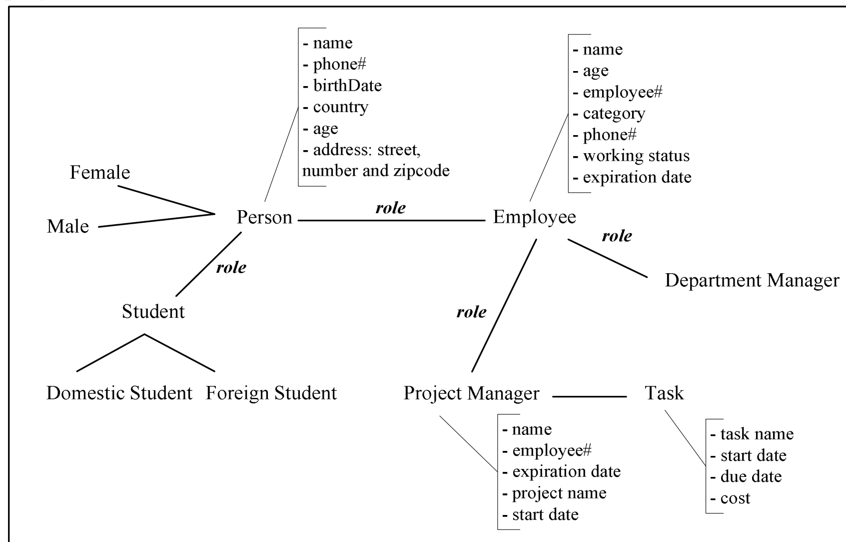

Fig. 1. Relationships involved in the example application

Despite its importance, the possibilities that conceptual modelling languages offer to deal with roles are very limited and cover only a very small part of their features. For instance, the ER model [6] considers roles as named places in a relationship; UML [29] considers that a role is an association end; in Description Logics [2] roles only denote binary relationships between individuals; in Nijssen's information analysis method (NIAM) [25] and in its descendants as Object-Role Modeling (ORM) [19] each fact type (relationship) involves a number of roles, hence, roles are named placed in the relationships.

Taking into account the complexity of the notion of role and the lack of support for roles in present conceptual modelling languages, it is clear that patterns to define such a common construct are needed in conceptual modelling.

2.3 Forces and Context

To account for the complexity of the notion of roles and variety of semantics found in them, we describe below the set of features that roles should meet, most of which have been identified by Steinmann [38]. In our case, these features are the forces that influence and should be resolved by the pattern.

We describe them by using some examples related to the scenario introduced above:

1. Ownership. A role comes with its own properties and behaviour, i.e., an instance of *Employee* has its own properties which may be different from the ones of the entity type that plays such a role.
2. Dependency. An instance of a role is related to a unique instance of its entity type and its existence depends on the entity type to which it is associated to, i.e., it is not possible to have an instance of *Student* not related to an instance of *Person.* Although a fundamental characteristic, there exist proposals considering that a role should remain unconnected to any entity type, for instance, to model the salary of a vacant position for department manager [36]. This work does not address such possibility.
3. Diversity. An entity may play different roles simultaneously, i.e., an instance of *Person* may play simultaneously the role of *Student* and *Employee*.
4. Multiplicity. An instance of an entity type may play several instances of the same role type at the same time. For instance, a person that registers to more than one university has multiple instances of *Student* related to it.
5. Dynamicity. An entity may acquire and relinquish roles dynamically, i.e., a person may become a student, after some years become an employee, finish his/her studies, become a project manager, start another program and so forth.
6. Control. The sequence in which roles may be acquired and relinquished can be subject to restrictions, i.e., a person may not become an employee after he/she has retired or when he/she is also studying two degrees. Note that this does not prevents an employee from studying two degrees in the future. The restriction needs to be true only when hiring the employee.
7. Roles can play roles. This mirrors that an instance of *Person* can play the role of *Employee* and an instance of *Employee* can also play de role of *ProjectManager*.

8. Role identity. Each instance of a role has its own role identifier, which is different from that of all other instances of the entity to which is associated with. This solves the so-called *counting problem* introduced by Wieringa et al in [44]. It refers to the fact that we need to distinguish the instances of the roles from the instances of the entity types that play them. For example, if we want to count the number of people that are students in a university (i.e., every person who is registered to at least a program in such university), the total number is less than the number of registered students in such university (in this case a person is counted twice if he or she is registered at two programs).
9. Adoption. Roles do not inherit properties from their entity types. Instead, instances of roles have access to some properties of their corresponding entities i.e., *Student* may adopt *name* and *address* properties of *Person* but neither *religion* nor *marital status* properties. Therefore, the *Student* role cannot use the last two referred properties.
10. Relationship independency. A role is meaningful even out of the context of a relationship. E.g., a person may play the role of student or employee without necessarily being tied to a university or a company respectively.
11. Common role for unrelated types. A set of unrelated types may play the same role [3]. For instance, a project manager may be the role of both employee and external service provider.
12. Sharing structure and behavior. Roles may have some common structure and behavior. For instance, the constraint that Maria may not become an employee before she is 16 years old should apply also to project manager.

2.4 *Roles as Participant Names* Pattern

2.4.1 Solution

A role is merely represented as a name assigned to the participation of an entity type in a relationship type. Although a rather limited representation, it is what conceptual modelling languages usually consider.

Figure 2 models the running example when considering roles as the participant names of a relationship type. Note that *ProjectManager* and *DepartmentManager* do not appear in the conceptual schema, since, in this approach, a role cannot play other roles. Students can neither be classified in domestic or foreign students.

2.4.2 Consequences

Only a small subset of the previous features is covered by the pattern. We justify each of them as follows:

2. Dependency: a relationship in a relationship type is always related to an instance of the participant entity types. For example, a student may only be defined if an instance of the relationship type relating *Person* and *University* exists.

3. Diversity: an entity type can participate in many different relationship types.

4. Multiplicity: this feature is partially covered since an entity type can play several times the same role (i.e., it may participate in the same relationship type) by providing the value of the multiplicy of the participant greater than one. However, two

relationships of a relationship type may not exist between the same participants (i.e a person cannot study twice in the same University).

5. Dynamicity: relationships can be added and deleted at any time.

Due to the limited expressiveness of participant names, the pattern does not support the following features:

1. Ownership: the participation of an entity type in a relationship type does not have properties nor behaviour

6. Control: we cannot attach constraints to the participation apart from multiplicity constraints.

7. Roles can play roles: only entity types can participate in other relationship types. Therefore, we cannot define that the role *Employee* has the role of *ProjectManager*.

8. Role identity: a relationship type has no identity, it is identified through its participants.

9. Adoption: by navigating through the relationship type where the role is defined we can access to all properties of the entity type. We cannot restrict the access to the personal number of person when navigating from employee.

10. Relationship independency: obviously, roles represented as participant names only make sense in the context of a relationship type.

11. Common role for unrelated types: the same role cannot be used in different relationship types.

12. Different roles may share structure and behaviour: since roles as participant names have no structure or behaviour they cannot share it.

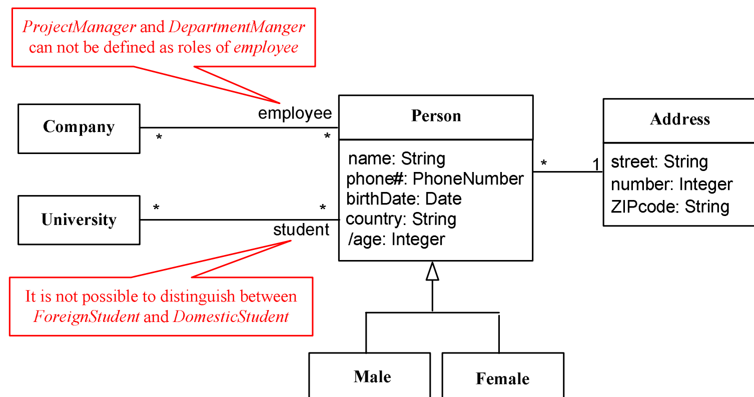

Fig. 2. Example of the *Roles as Participant Names* Pattern

2.4.3 Design and implementation

The design and implementation of roles defined following this pattern in object oriented languages is straightforward. Roles are simply transformed into attributes of the entity type. The multiplicity and type of these attributes is obtained from the definition of the relationship type including the role.

2.4.4 Known Uses

This pattern is useful when we want to qualify the participation of an entity type in a relationship type but we are not interested in defining properties or behaviour of that participation. For instance, if we are only interested on a person becoming a student without worrying about his or her properties as student, like the degree he/she is studying.

2.5 *Roles as Subtypes* Pattern

2.5.1 Solution

Role entity types are represented as subtypes of the entity type playing them. For instance, *Student* and *Employee* would appear as subtypes of *Person*. Quite obviously, such a solution requires dynamic and multiple classification, since a person can change his/her role and play several roles simultaneously.

As an example Figure 3 shows the running example when considering roles as subtypes.

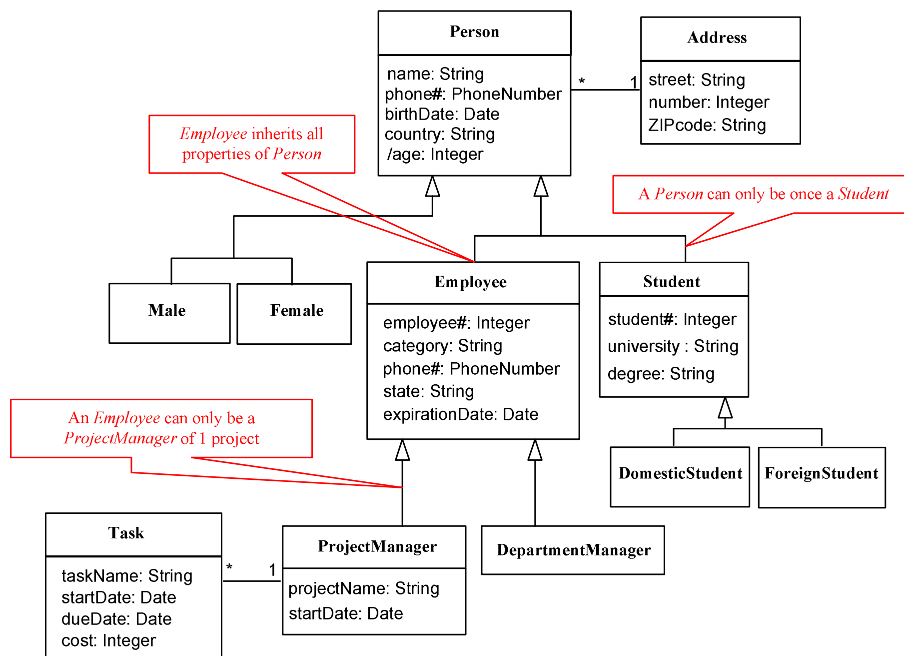

Fig. 3. Example of the *Roles as Subtypes* Pattern

2.5.2 Consequences

The following list of role features is supported for this pattern:

1. Ownership: as a subtype, the role can define its own properties and behaviour.
2. Dependency: all subtype instances are instances of their supertypes.

3. Diversity: by assuming multiple classification, an entity may appear in several role subtypes (i.e., a person may be an employee and a student at the same time).
5. Dynamicity: by assuming dynamic classification role instances can be added and removed from the subyptes at any time.
7. Roles can play roles: this feature is simulated by defining a role type as a subtype of another role type. For instance, *ProjectManager* is defined as a subtype of *Employee*.
10. Relationship independency: as entity types, roles have their own existence independent from any relationship type.
12. Different roles may share structure and behaviour: by assuming multiple inheritance, two roles may share structure and behaviour if they inherit them from a common supertype.

On the other hand some important features are not covered by the pattern:
4. Multiplicity: since Employee is a subtype of *Person*, we cannot define that a person plays simultaneously twice the role of *Employee*.
6. Control: as entity types, we can define constraints on role types. However, the constraints are static and thus must be satisfied at any time. We cannot define constraints that only apply when the role is inserted or deleted.
8. Role identity: as a subtype, the identifier of the role instance is the same as the identifier of the entity type instance. Therefore, the counting problem mentioned before is not solved. In fact, the counting problem is not solved either because multiplicity cannot be addressed with this pattern.
9. Adoption: with specialization we cannot restrict which attributes are adopted by the roles because they inherit all the attributes and relationships of their supertype (i.e. *Employee* inherits all the *Person*'s attributes).
11. Common role for unrelated types: this could be simulated using multiple inheritance. However, in such a case the role would inherit the properties of all its supertypes which is not what we meant. For instance, if we need *ProjectManager* to be the role of both *Employee* and *ExternalServiceProvider*, we could define *ProjectManager* as a subtype of both *Employee* and *ExternalServiceProvider* but then *ProjectManager* would inherit the properties of *Employee* as well as the properties of *ExternalServiceProvider*.

2.5.3 Design

The design and implementation of roles defined following this pattern in object oriented languages is not straightforward, since, in general, they do not support multiple nor dynamic classification. Fowler [13] and Pelechano et al. [34] comment some design patterns to cope with these issues.

2.5.4 Known Uses

Probably, this is the most intuitive way to represent roles. Provided that the previous limitations are not a problem in the specific application, this pattern represents a good combination between simplicity and expressiveness. However, we should also take into account that the design of this pattern is not trivial.

2.6 *Roles as Interfaces* Pattern

2.6.1 Solution

Roles are represented as interfaces. An interface represents a declaration of a set of coherent public features and obligations [29]. To define that an entity type plays a certain role we must specify that the type realizes (i.e., implements) the interface corresponding to that role.

The implementation relationship signifies that the entity type conforms to the contract specified by the interface (i.e., the entity type provides an operation for every operation defined in the interface and a property for each feature).

Figure 4 shows the running example when considering roles as interfaces. For instance, to specify that *Person* plays the role of *Employee*, *Employee* is defined as an interface that is implemented by *Person*. Note that this obliges *Person* to contain all the attributes defined in *Employee*, since an entity needs to contain all attributes of all its interfaces. Students are not classified in domestic or foreign since the notion of subtypes cannot be applied to interfaces. We could define them as interfaces extending the *Student* interface but the resulting semantics would be different (for example, the state of a person being a student would be completely independent from the state of a person being a domestic student, since interfaces inherit only the specification of extended interfaces but not its state).

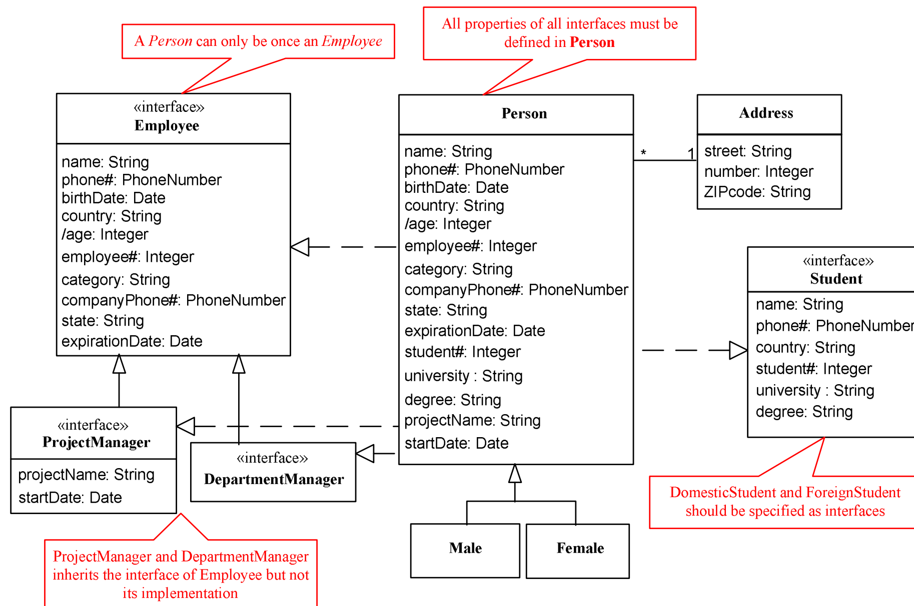

Fig. 4. Example of the *Roles as Interfaces* Pattern

2.6.2 Consequences

The features covered for this pattern are:

2. Dependency: the role does not exist as a separate instance from the instance of the entity type.
3. Diversity: an entity type can implement several interfaces.
10. Relationship independency: as interfaces, roles do not depend on any relationship of the entity type.
11. Common role for unrelated types: interfaces can be implemeted by many unrelated types.
12. Different roles may share structure and behaviour: this can be simulated by defining inheritance relationships between interfaces.

The features not covered for this pattern are:
1. Ownership: even though an interface defines its own properties, it does not have an independent state from the state of the instances of the entity type implementing them.
4. Multiplicity: an entity type cannot implement twice an interface, and thus, a entity type can only play once the same role.
5. Dynamicity: every instance of the entity type plays all roles corresponding to the interfaces implemented by the type during its whole life. All interfaces are acquired when the instance is created.
6. Control: since all roles are acquired at the beginning, this property does not make sense.
7. Roles can play roles: interfaces cannot implement other interfaces. For instance, *Employee* cannot implement the *ProjectManager* interface. When defining *ProjectManager* we can extend the specification of *Employee* but the semantics are the same as if two independent interfaces were defined since it is *Person* and not *Employee* who implements the interface *ProjectManager*.
8. Role identity: interfaces do not have instances.
9. Adoption: interfaces do not adopt any of the properties of the entity type. On the contrary, each interface defines its independent set of properties that must be implemented by the entity type.

2.6.3 Design and implementation

One of the advantages of this pattern is that the design and implementation of roles as interfaces is a direct translation from the conceptual schema since most object oriented languages perfectly support the notion of interfaces.

2.6.4 Known uses

Despite its major drawbacks, the main usage of this pattern is the definition of conceptual schemas where roles must be played by unrelated entity types. Also, as Fowler in [13] proposes, the pattern can be used to integrate seamlessly the specification of CS with its implementation when the implementation language does not support multiple and dynamic classification.

2.7 *Roles as Reified Entity Types* Pattern

2.7.1 Solution

Roles are represented as reified entity types of a relationship type between the entity type playing the role and another entity type (called the *companion* entity type). For instance, the student role can be defined as the reified entity type of the relationship between *Person* and *University* (although a more appropriate name for the reified type could be enrolment). Note that it is not clear whether *Student* is a role of *Person* or *University*.

Choosing the right companion entity type is not easy. In fact, sometimes it does not exist and must be created specifically to be able to define the role. For instance, to define the student role we have to include in the CS the *University* type. Note that, depending on the purpose of the CS, we may not be interested in recording data about the universities where people study. In such a case, a plain string recording the name of the university as an attribute of the student could be enough (as in the roles as subtypes approach).

On the other hand, the election may impose some constraints to the role since there could not exist two relationships of a relationship type between the same participants. In the previous example, choosing *University* as companion entity type prevents a person to study twice in the same university.

Figure 5 shows the running example when considering roles as reified entity types.

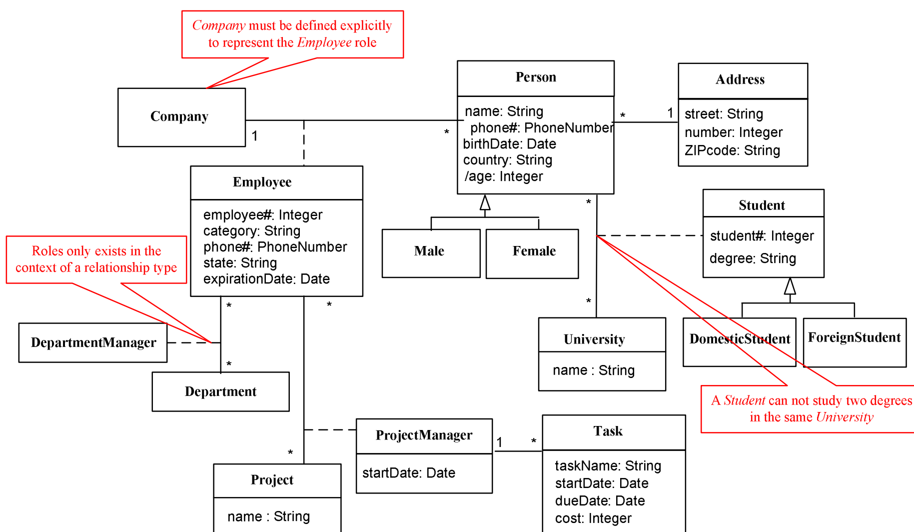

Fig. 5. Example of the *Roles as Reified Entity Types* Pattern.

2.7.2 Consequences

The features covered for this pattern are:

1. Ownership: as an entity type, the role can define its own properties and behaviour.
2. Dependency: as a reified entity type, an instance of the role type can only exist when the participants of the relationship also exist.
3. Diversity: an entity type may participate in several relationships.
5. Dynamicity: instances of reified entity types can be inserted and deleted at any time.
7. Roles can play roles: a reified entity type can be a participant in other relationship types. For instance, the reified entity type Employee is one of the participants of the relationship defined to specify the role DepartmentManager
8. Role identity: the role has its own identity as a reified entity type, different from the identity of the entity type.
12. Different roles may share structure and behaviour: this can be simulated by means of generalization and specialization relationships between the reified types.

On the other hand, the roles do not cover:

4. Multiplicity: it is restricted by the choosen companion class.
6. Control: as entity types, we can define constraints on role types. However, we cannot define constraints that only apply when the role is inserted or deleted.
9. Adoption: we cannot restrict which attributes are adopted by the roles since, by default, all properties can be acceded.
10. Relationship independency: as reified entity types, roles can only exist in the context of a relationship type.
11. Common role for unrelated types: two different relationship types cannot share the same reified entity type.

2.7.3 Design and implementation

The reified entity types are designed and implemented as classes with a set of single-valued additional attributes, which refer to each of the participants of the relationship type.

2.7.4 Known Uses

This pattern can be regarded as an extension of the previous *roles as participant names* pattern. Therefore, this pattern is useful to qualify the participation of an entity type in a relationship type. Moreover, by using the pattern we can define properties and behaviour of that participation.

As a disadvantage, apart from the features not covered by the pattern, the use of this pattern may increase the complexity of the resulting CS since for each role we must create a relationship type, a reified relationship type and, when not present in the CS, the companion entity type.

3 *Roles as EntityTypes* Pattern

Previous patterns cover only a subset of the required role features. We propose to use our *Role as Entity types* pattern when the full expressivity of the role concept is needed. This pattern is an updated and extended version of the one presented in [5].

The basic aim of the pattern is to represent roles as an entity type related to the entity type playing the role by means of a special kind of relationship type, the *RoleOf* relationship type.

Next sections discuss in depth the pattern solution, its consequences, design and known uses.

3.1 Solution

We divide the solution of our role pattern in two subsections. The first one deals with the structural aspects of roles while the second one deals with their evolution.

3.1.1 Structural Aspects of Roles

We believe there is not a fundamental difference between roles and entity types since roles have their own properties and identity. Therefore, when using this pattern we represent roles as entity types with their own attributes, relationships and generalisation/specialisation hierarchies. For practical reasons, we call *role entity* types (or simply role if the context is clear) the entity types that represent roles and *natural entity types*[1] (or simply entity types) the entity types that may play those roles.

We define the relationship between a role entity type and its natural entity type by means of a new generic relationship type, the *RoleOf* relationship. A generic relationship type is a relationship type that may have several realizations in a domain [27]. Each realization of this generic relationship type is a specific relationship type relating a natural entity type to a role entity type to indicate that the natural entity type may play the role represented by the role entity type.

In the relationship type we also specify the properties (attributes and associations) of the natural entity type that are adopted by the role entity type.

Note that, since roles may play other roles, the same entity type may appear as a role entity type in a *RoleOf* relationship and as a natural entity type in a different *RoleOf* relationship.

Although this representation may be expressed in many conceptual modelling languages, in this work, we only adapt it to UML [29] and OCL [28]. See Olivé in [27] for a general discussion about the implementation of generic relationship types in conceptual schemas.

To represent the *RoleOf* relationship we use the standard extension mechanisms provided by UML, such as stereotypes, tags and constraints. Stereotypes allow us to define (virtual) new subclasses of metaclasses by adding some additional semantics and properties (tags) to its base entity type. A stereotype may also define additional

---

[1] The *natural entity type* of a role relationship has sometimes been called *object class* [8, 44] *ObjectWithRoles* [17], *natural type* [18, 38], *base class* [7, 31], *entity type* [1], *entity class* [3], *base role* [33], or *core object* [4].

constraints. It is worth to notice that merely using these lightweight[2] extension mechanisms ensures that the pattern can be easily integrated in UML conceptual schemas.

We represent the *RoleOf* generic relationship type by means of the «RoleOf» stereotype. The base class of the stereotype is the *Association* metaclass, which represents association relationships among classes. Each specific relationship type is labelled with this stereotype. The stereotype also permits the definition of the properties[3] the role adopts from the natural entity type. They are represented with a multivalued attribute, called *adoptedProperties*. Figure 6 shows the definition of the «RoleOf» stereotype.

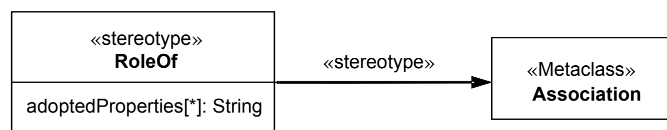

Fig. 6. Definition of the RoleOf stereotype

As an example, figure 7 shows the running example introduced in section 2.2 using this pattern. Note that all roles are represented as entity types with a «RoleOf» relationship type relating the role with is entity type. For instance, the role *Student* is represented as an entity type related to *Person* through a «RoleOf» relationship type. In the relationship type it is also indicated that student adopts the properties: *name*, *address*, *phone#* and *country* from *Person* (its natural entity type). *Employee* participates in three «RoleOf» relationship types, one as a role of *Person* and the other ones as a natural entity type playing the role of *ProjectManager* and *DepartmentManager*.

The stereotyped operations, also shown in the figure, will be further described in the following section.

To complete the definition of the static aspects of roles we must attach some constraints to the «RoleOf» stereotype in order to control the correctness of its use. There already exist proposals to automatically ensure the consistency of a conceptual schema according to the conceptual modelling language metamodel extended with stereotypes [41].

The constraints are the following:

- A stereotyped «RoleOf» association is a binary association with multiplicity '1' and settability *readOnly* in an association end.
- Each value of the *adoptedProperty* tag must coincide with the name of a property of the natural entity type.
- A role entity type can only be related throughout a *RoleOf* relationship to at most a natural entity type.
- No cycles of roles are permitted. A role entity type may not be related throughout a direct or indirect *RoleOf* relationship to itself.

[2] In contrast with heavyweight mechanisms that involve the creation of new metaentity types.
[3] A property in UML 2.0 [29] represents both the attributes and associations of an entity type.

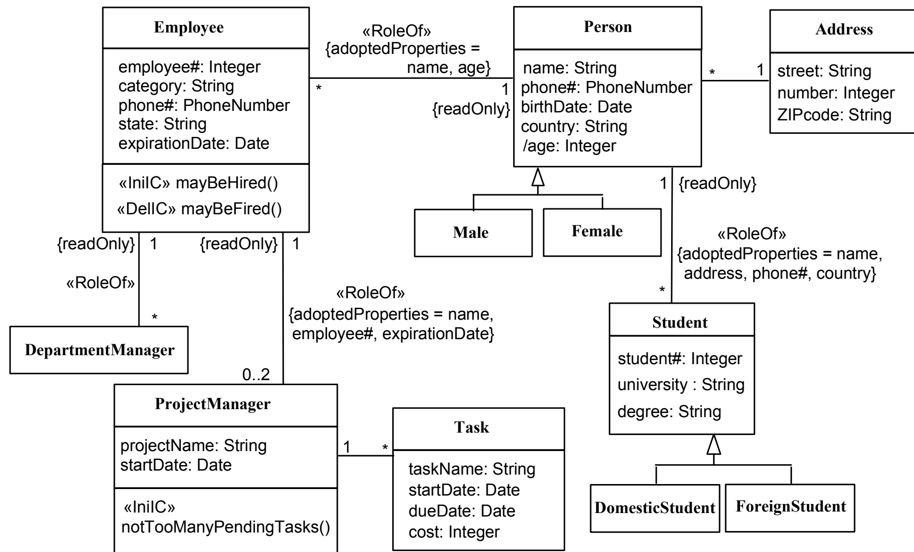

Fig. 7. Example of *RoleOf* relationships in the UML

Properties adopted by the role from its natural entity type may be considered as implicit properties of the role entity type. Nevertheless, in order to facilitate the use of these adopted properties (for instance, when writing OCL expressions) we may need to include them explicitly in the role entity type. In this case, we add an extra property in the role entity type for each adopted property. These extra properties are labeled with the «adopted» stereotype to distinguish them from the own properties of the role entity type. In addition, they are derived. Their derivation rule always follows the general form:

```
context RoleEntityType::adoptedPropertyX: Type
derive: naturalEntityType.propertyX
```

Note that, to facilitate the work of designers, these added properties can be automatically generated.

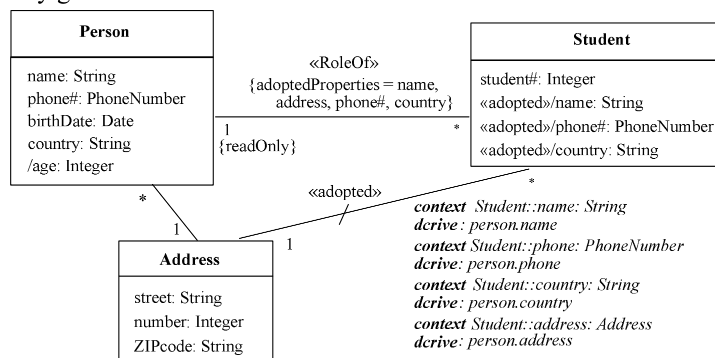

Fig. 8. Example of the *Student* role entity type

Figure 8 extends a subset of the previous example illustrating the *Student* role entity type including its adopted properties.

Likewise, when the CS includes operations role entity types can also adopt said operations. For instance, if we express the *age* derived property of the *Person* entity type as a query operation (Figure 9) we may be interested in defining that the *age* operation can also be executed over employees (indicated in the *adoptedOperation* tag). Operations are adopted following a delegation mechanism, i.e., the body of the adopted operation delegates the execution to the original operation.

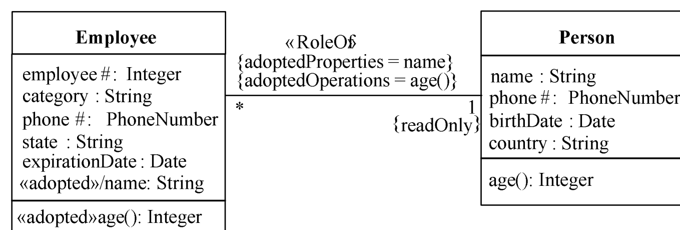

Fig. 9. Adoption of the *age* operation

For instance, the adopted *age* operation in the *Employee* role entity type would be defined as:

```
context Employee::Age():Natural
body: person.age()
```

3.1.2 Role Acquisition and Relinquishment

So far, we have introduced a representation of the static part of the *Roles as Entity Types* Pattern. Nevertheless, this is not enough since role instances may be added or removed dynamically from an entity during its lifecycle and this addition or removal may be subjected to user-defined restrictions.

Since roles are represented as entity types we may define constraints on roles in the same way we define constraints on entity types. Some of the constraints are inherent to our role representation (for example, that a person must play the role of *Employee* to play the role of *ProjectManager*, is already enforced by the schema). Other restrictions may be expressed by means of the predefined constraints of the UML. For example, to restrict that an *Employee* cannot play more than twice the *ProjectManager* role simultaneously, it is enough to define a cardinality constraint in the relationship type. The definition of the rest of constraints requires the use of a general-purpose language, commonly OCL in the case of UML. For instance, we could specify OCL constraints to control that:

- A *Person* can only play the role of *Employee* if he/she is between 18 and 65 years old:

```
context Employee
inv: self.age>=18 and self.age<=65
```

- Any task of a ProjectManager must finish before his contract expires

```
context Task
inv:self.dueDate<self.projectManager.expirationDate
```

These OCL constraints are static, and thus, the role instances must satisfy them at any time. However, many of the restrictions that may be involved in the evolution of roles only apply at particular times, particularly they only need to be satisfied when the role is acquired or when it is relinquished. To specify such constraints we use the notion of creation-time constraints defined by Olivé in [26] and, in a similar way, we define the deletion-time constraints.

Creation-time constraints must hold when the instances of some entity type are created (in our case when the role is created). Deletion-time constraints must hold when the instances of some entity type are deleted (in our case when the role is deleted). These constraints are represented as operations, also called *constraint* operations, attached to the entity types and identified by a special stereotype. The creation-time constraint operations are marked with the stereotype «IniIC». We define the stereotype «DelIC» for the deletion-time constraint operations.

These operations return a boolean that must be true to indicate that the constraint is satisfied. If the operation returns false (i.e., the constraint is not satisfied) then the creation or deletion event of the role is not accomplished. When appropriate, the operations are automatically executed by the information system.

As an example, the constraints in Figure 7 can be defined as follows:

- A person cannot become an employee if he/she is studying two university programs simultaneously. Note that this does not imply that a person that is already an employee may not apply for two degrees.

```
context Employee :: mayBeHired () : Boolean
  body:  self.person.student->size()<2
```

- An employee may not be fired if he or she is in maternity leave.

```
context Employee :: mayBeFired () : Boolean
  body: self.workingStatus<>'MaternityLeave'
```

- An employee may not become a new project manager if he/she still holds more than ten pending tasks.

```
context ProjectManager::notTooManyPendingTasks():
                                          Boolean
 body : self.employee.projectManager.tasks ->
          select(dueDate>Today)->size()<=10
```

3.2 Consequences

The pattern achieves most of the role features outlined before:

1. Ownership. As roles are represented as entity types, they may have their own properties.

2. Dependency. The cardinality '1' with the tag {readOnly} ensures that all role instances depend on a unique instance of the natural entity type.

3. Diversity. Entity types may have many *RoleOf* relationships.

4. Multiplicity. This is obtained by defining a cardinality greater than one in the *RoleOf* relationship.

5. Dinamicity. Entities are related to their roles through an association. Thus, an entity may acquire or retract instances of a role at any time.

6. Control. The sequence in which roles may be acquired and relinquished can be subjected to restrictions, including creation-time and deletion-time constraints.

7. Roles can play roles. Roles are represented by ordinary entity types. So, they can be participants of a *RoleOf* relationship.

8. Role identity. As roles are represented as entity types, their instances have their own identifier.

9. Adoption. The *adoptedProperty* tag of the *RoleOf* relationship allows the definition of the adopted properties.

10. Relationship independency. As entity types, roles are independent from relationship types.

12. Different roles may share structure and behavior. As entity types we can define generalization relationships between roles.

As a trade-off, our pattern does not directly supports the remaining feature (11. Common role for unrelated types). However it can be easily represented. For instance, if we need *ProjectManager* to be the role of both *Employee* and *ExternalServiceProvider*, we could define a common supertype for *Employee* (understood as *InternalServiceProvider*) and *ExternalServiceProvider*, called *ServiceProvider*, which plays the role of *ProjectManager*.

An alternative is to define two different RoleOf relationship types, one between *ProjectManager* and *Employee* and another one between *ProjectManager* and *ExternalServiceProvider*. Both relationship types are specified with a *xor* constraint to prevent a project manager being an employee and an external service provider at the same time. *ServiceProvider* is not needed. On the other hand, the management of the adopted properties is more complex.

Figure 10 shows the example using both alternatives.

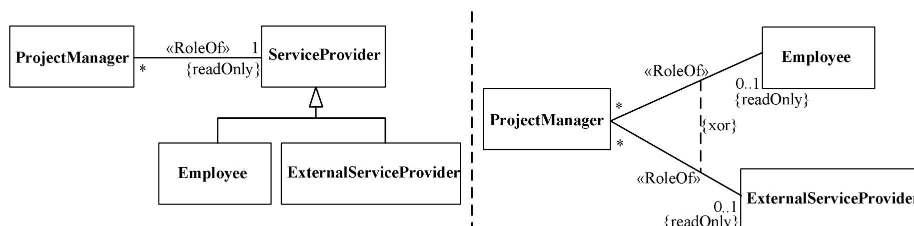

Fig. 10. Representing *common role for unrelated types* feature

3.3 Design and Implementation

There are some design patterns useful for designing and implementing roles in object oriented languages [13]. However, most of them are unable to deal with our proposed role semantics completely. A well-known pattern close to our role defined semantics is the *Role Object Pattern* [4]. This pattern is especially well suited for role implementation when roles are deemed as a specialization (or a kind of specialization) of its entity type (see Pelechano et al. in [33] as an example).

Nevertheless, this pattern is not entirely appropriate for designing our conceptual modelling pattern. We encounter two main problems in the *Role Object Pattern*. First, it uses a common superclass for all the roles of the entity type. In our approach, roles are independent entity types with not necessarily any common properties that justify this superclass. Second, all the roles are forced to have the same inherited properties; it is not possible to define different adopted properties for each role.

This is the reason why we advocate here for an adapted version of this pattern that takes into account our complete role semantics, including the adoption mechanism and the creation-time and deletion-time constraints.

Given a natural entity type and the set of its roles, we create a class for the natural entity type and a class for each role. We create a different relationship between the natural entity type and each of its roles. This relationship will be used to navigate from the natural entity type to its roles and vice versa. We add to the natural entity type two new operations *addRole* and *deleteRole* in charge of adding (deleting) roles to the natural entity after checking the creation-time (deletion-time) constraints. We could also add other useful operations when dealing with roles, such as *hasRole* (to check whether an entity plays a role) or *getRole* (to obtain a role played by the entity).

The problem of the design of the adopted properties may be regarded as the same problem as designing derived information. In general, from a design and/or implementation point of view, there are two different approaches to deal with derived information. The attributes may be computed if they are calculated by means of an operation or may be materialized if they are explicitly stored in the class. In this case, for each adopted property we add an extra operation to the role class that returns the value of the property of the natural entity type. The operation accesses the property of the natural entity type navigating through the relationship.

Figure 11 summarizes our proposal.

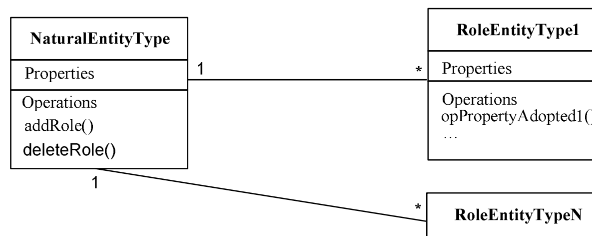

Fig. 11. Summarized class diagram of the design

In figure 12, we apply the proposed design pattern to a part of the conceptual schema of figure 7. Note that *Employee* is both a role for the *Person* entity type and a natural entity type for the *ProjectManager* role, and thus, it presents both a reference to *Person* (as a role entity type) and the operations *addRole* and *deleteRole* (as a natural entity type). Additionally, *Employee* includes also the *name* and *age* operations to get this information from *Person*.

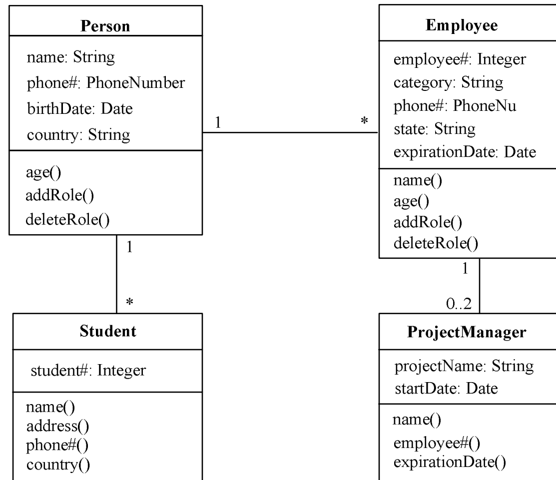

Fig. 12. Example of an application of the design

This structure can be directly implemented in any common object-oriented language. An example of the implementation in the Java Language can be found in Appendix A.

3.4 Known Uses

This pattern should be used to represent the full expressiveness of roles in conceptual schemas.

In contrast to other approaches where the complexity of the CS is really increased when using roles due to the special construct needed to represent them, our pattern allows a plain integration of roles in CS. Therefore, there are no trade-offs that prevent from applying the pattern whenever it may be useful.

4 Related Work

The role concept has been widely addressed in the literature. Although all approaches present their own characteristics, they can be grouped in four basic approaches to represent roles: 1 - roles as the name of a participant in a relationship type [6, 10, 19, 29]; 2 - roles as a sort of subtypes or supertypes of the natural entity types [1, 31, 37];

3 - roles as interfaces [21, 23, 40]; and 4 - roles as a distinct element from an entity type but coupled to it [3, 7, 8, 11, 17, 20, 22, 35, 38, 42 , 44, 45].

The first three families are similar to our *Roles as Participant Names*, *Roles as Subtypes* and *Roles as Interfaces* patterns, respectively. Therefore, the major advantages and drawbacks of these three groups are mainly the same commented for the corresponding patterns in Section 2.

In this section we focus on the comparison between our *Roles as Entity types* pattern and the other approaches also considering a role as a distinct element from an entity type but coupled to it.

Table 1 compares the most representative approaches in terms of the role features they can handle. Most of these approaches use different semantics from the ones presented in this paper or are unable to handle the full role semantics.

All approaches shown in the table fulfil the ownership, dependency, diversity and dynamicity features.

Table 1. Comparison of role representation approaches

Features \ Approaches	Bachman and Daya [3]	Chu and Zhang [7]	Dahchour, Priotte and Zimányi [8]	Fan, Barker, Porter and Clark [11]	Gottlob, Schrefl and Röck [17]	Jodlowski, Habela, Podzien and Subieta [20]	Kristensen [22]	Pernici [35]	Steimann (DKE) [38]	Thalheim [42]	Wieringa, de Jonge and Spruit [44]	Wong, Chau and Lochovsky [45]
Ownership	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Dependency	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Diversity	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Multiplicity	✕	✕	✔	✕	✔	✔	✔	✔	✔	✔	✔	✔
Dynamicity	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Control	✕	✕	✔	✕	✕	✕	✕	~	✕	✕	~	✕
Roles can play roles	✕	✔	✔	✕	✔	✔	✔	✕	✕	✔	✔	✔
Identity	✕	✔	✔	✕	~	✔	✕	✕	✕	✔	✔	~
Adoption	✔	✕	✕	✕	✕	✕	✕	✕	✕	✕	✕	✕
Relantionship Independency	✔	✕	✔	✕	✔	✔	✔	✔	✕	✔	✔	✔
Common role for unrelated types	✔	✕	✕	✔	✕	✕	✔	✕	✔	✕	✕	✔
Sharing structure and behavior	✕	✔	✔	✔	✔	✔	✔	✕	✔	✕	✔	✔

However, few approaches consider roles with their other identity (thus, solving the counting problem). Some of them propose alternative techniques to distinguish between different role instances of the same natural instance. For instance, Gottlob et al [17] mixes the identifier of the natural instance with the value of a special attribute, called qualifier and Wong et al. [45] uses the state of the role instance.

Even more critical is the support of the control and adoption features.

Most of them do not handle the control feature. Some allow the definition of static constraints. Additionally, Pernici [35] and Wieringa [44] take into account the sequence in which roles are acquired and relinquinshed, but do not consider the definition of additional restrictions over the sequence (as our creation-time and deletion-time constraints).

Adoption is neither supported. Most approaches define that roles can (or cannot) access all the properties of the natural entity type but they do not provide a mechanism to indicate which properties may be adopted.

Our alternative suggesting roles as separated entity types fulfils the role semantics.

We believe one of the main advantages of our *Roles as Entity Types* pattern over previous approaches is that we handle the complexity of role semantics in a very simple manner since we represent roles and their evolution with already existing elements (entity types and constraints) without adding completely new language constructs (as done by several of the previous approaches). Therefore, the designer can easily use the patterns to specify roles in conceptual schemas. In addition, our pattern describes a representation of roles in the standard UML, and thus, the pattern can be directly incorporated into current UML CASE tools.

We would also like to remark that our pattern is complete and feasible in the sense that it includes the design and the implementation of the pattern, in contrast to most of previous approaches that do not state how this could be achieved.

5 Conclusions

This paper identifies the most important features of roles and presents a set of conceptual modelling patterns to facilitate the representation of roles in conceptual schemas. Each pattern is characterized in terms of the features it covers. We also review their design and implementation.

Roles as Entity Types pattern is of special importance. We propose using this pattern when we need to represent the full expressivity of roles in CSs. We have adapted the pattern to the UML conceptual schemas. To our knowledge, ours is the first UML standard extension that defines roles in conceptual schemas specified with this language. Because of its simplicity, the pattern can be easily implemented in any CASE tool in order to allow designers the use of the role concept.

The pattern includes the static aspects of roles as well as their evolution. We define roles as entity types (role entity types) related to natural entity types by means of a generic *RoleOf* relationship type that includes the adoption of properties from the natural entity types by the role entity types. We have extended UML by means of the *«RoleOf»* stereotype to be able to represent such kind of relationships. To specify the role evolution, we use two special kinds of constraints: creation-time constraints and deletion-time constraints.

It would be interesting to semi-automate the selection and application of these patterns in CS. Given the set of role features the designer needs to take into account, the CS and a set of roles, we could integrate the roles in the CS by using the simplest pattern covering the required role features. Additionally, given the CS with the roles included, we would like to automate its design and implementation by means of an

application that, given a conceptual schema (for instance, represented in XMI [30]), generates automatically the corresponding classes in the target object oriented language. These are directions in which we plan to continue our work.

Acknowledgements

We would like to thank Jordi Conesa, Dolors Costal, Xavier de Palol, Cristina Gómez, Antoni Olivé, Anna Queralt, Maria Ribera Sancho, Ernest Teniente for their many useful comments in the preparation of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIC2002-00744.

References

1. A. Albano, R. Bergamini, G. Ghelli, R. Orsini, "An Object Data Model with Roles", Proceedings of the 19th Very Large Data Bases (VLDB) Conference. Morgan Kaufmann, 1993, pp. 39-51.
2. F. Baader, W. Nutt, "Basic Description Logics", In: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors, The Description Logic Handbook: Theory, Implementation, and Applications. CambridgeUniversity Press, 2003
3. C.W. Bachman, M. Daya. "The Role Concept in Data Models", Proceedings of the 3rd Very Large Data Bases (VLDB) Conference, 1977, pp. 464-476.
4. D. Bäumer, D. Riehle, W. Wiberski, M. Wulf. "The Role Object Pattern", Proceedings of Pattern Languages of Programming (PLoP) Conference 1997. Technical Report WUCS-97-34. Washington University Dept.
5. J. Cabot, R. Raventos, "Roles as Entity Types: A Conceptual Modelling Pattern", Proceedings of the 23rd International Conferece on Conceptual Modeling (ER'04), LNCS 3288, Springer, pp. 69-82
6. P.P. Chen. "The entity-relationship model: Towards a unified view of data, ACM Transactions on DatabaseSystems 1 (1), 1976, pp. 9-36.
7. W.W. Chu, G. Zhang, "Associations and Roles in Object-oriented Modeling", Proceedings of the 16th International Conferece on Conceptual Modeling (ER'97), LNCS 1331, Springer, pp. 257-270.
8. M. Dahchour, A. Pirotte, E. Zimányi, "A role model and its metaclass implementation", Information Systems, 29 (2004) pp. 235-270.
9. R.Depke, G.Engels, J.M. Küster, "On the Integration of Roles in the UML", Technical Report No. 214, University of Paderborn, August 2000.
10. E. Falkenberg, "Concepts for modelling information", Proceedings of the IFIP Conference on Modelling in Data Base Management Systems, North-Holland, Amsterdam; 1976, pp. 95-109.
11. J. Fan, K. Barker, B.W. Porter, P. Clark, "Representing roles and purpose", Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001), pp. 38-43.
12. E. B. Fernandez; X. Yuan. "Semantic Analisis Patterns", Proceedings of the 19th Int. Conference on Conceptual Modeling (ER'00), LNCS 1920, Springer 2000, pp. 183-195.

13. M. Fowler, “Dealing with Roles”, Pattern Languages of Programming (PLoP '97) and EuroPLoP '97 Conference, Technical Report #wucs-97-34, Dept. of Computer Science, Washington University, 1997.
14. M. Fowler, “Analysis Patterns: Reusable Object Models”, Addison-Wesley, 1997.
15. E.Gamma, R.Helm, R.Johnson, J. Vlissides, “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994.
16. A. Geyer-Schulz, M. Hahsler, “Software Reuse with Analysis Patterns”, Proceedings of the 8th Americas Conference on Information Systems (AMCIS 2002), August 2002, pp. 1156-1165.
17. G. Gottlob, M. Schrefl, B. Röck, “Extending Object-oriented Systems with Roles”, ACM Transactions on Information Systems 14 (3), 1996, pp. 268-296.
18. N. Guarino, “Concepts, Attributes and Arbitrary Relations”, Data & Knowledge Engineering 8, 1992, pp. 249-261.
19. T. Halpin, “Conceptual Schema & Relational Database Design”, Second Edition, Prentice-Hall of Australia Pty Ltd: Sydnes, 1995
20. A. Jodłowski, P. Habela, J. Płodzien, C. Subieta, “Extending OO Metamodels towards Dynamic Object Roles”, R. Meersman et al. (Eds.): On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, LNCS 2888 Springer 2003, pp. 1032–1047.
21. E.A. Kendall, Role Modeling for Agent System Analysis, Design, and Implementation, IEEE Concurrency, vol. 8 no. 2, 2000, pp. 34-41.
22. B.B. Kristensen, Object Oriented Modeling with Roles, Proceedings of the 2nd International Conference on Object-Oriented Information Systems (OOIS’95), 1995
23. D. Lea, J. Marlowe, “Interface-Based Protocol Specification of Open Systems using PSL”, 9th European Conference ECOOP'95 - Object-Oriented Programming, LNCS 952 Springer 1995, pp. 374-398.
24. F.G. Mossé, “Modeling Roles - A Practical Series of Analysis Patterns”, Journal of Object Technology (JOT) , vol.1 no.4, 2002, pp.27-37.
25. G.M. Nijssen and T.A. Halpin. “Conceptual Schema and Relational Database Design: a fact oriented approach”. Prentice-Hall, Sydney, Australia, 1989.
26. A. Olivé, “Integrity Constraints Definition in Object–Oriented Conceptual Modeling Languages”, Proceedings of the 22th International Conference on Conceptual Modeling (ER’03), LNCS 2813, 2003, pp.349-362.
27. A. Olivé, “Representation of Generic Relationship Types in Conceptual Modeling·, Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE’02), LNCS 2348, pp. 675-691
28. OMG, “UML 2.0 OCL Specification”, Adopted Specification (ptc/03-10-14), 2003
29. OMG, “UML 2.0 Superstructure Specification”, Adopted Specification (ptc/03-08-02), 2003
30. OMG, “OMG XML Metadata Interchange Specification”, v.1.2, January 2002.
31. M.P. Papazoglou, B.J. Krämer, “A database model for object dynamics”, The Very Large Databases (VLDB) Journal (6), January 1997, pp. 73-96.
32. M. P. Papazoglou, “Modeling Object Dynamics”, in. M.P. Papazoglou, S. Spaccapietra, Z.Tari (Eds.), Advances in Object-Oriented Data Modeling. MIT Press 2000, pp. 195-217.
33. V. Pelechano, M. Albert, E. Campos, O. Pastor, “Automating the Code Generation of Role Classes in OO Conceptual Schemas”, Proceedings of the 4st International Conference on Enterprise Information Systems (ICEIS 2002), 2002, pp. 656-686.
34. V. Pelechano, O. Pastor, E. Insfrán, “Automated code generation of dynamic specializations: an approach based on design patterns and formal techniques”, Data & Knowledge Engineering 40, 2002, pp. 315-353

35. B. Pernici, "Objects with Roles", Proceedings of the Conference on Office Information Systems, SIGOIS Bulletin, vol. 11, no. 2/3, ACM Press, New York, 1990, pp. 205-215.
36. T. Reenskaug, P.Wold, O.A. Lehne, Working with Objects: The OOram Software Engineering Method, Prentice-Hall, Englewood Cliffs, NJ, 1995.
37. J. Sowa, "Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley Publishing Company, New York, 1984.
38. F. Steimann, "On the Representation of Roles in Object-oriented and Conceptual Modelling", Data & Knowledge Engineering 35, 2000, pp. 83-106.
39. F. Steimann, "A Radical Revision of UML's Role Concept", UML 2000: The Unified Modelling Language, LNCS 1939, Springer, pp. 194-209.
40. F. Steimann, "Role=Interface", Journal of Object-Oriented Programming, October/November 2001, Vol. 14, Num. 14, pp. 23-32.
41. J.G. Süß, A. Leicher, F. Chabarek, "Software Model Engineering and Reuse with the Evolution and Validation Environment", N.Guelfi, E. Astesiano, G. Reggio (Eds.): Scientific Engineering of Distributed Java Applications, Third International Workshop, FIDJI 2003, November 27-28, 2003, Revised Papers, LNCS 2952, Springer 2004, pp. 196-105.
42. B. Thalheim, "Entity-Relationship Modeling: Foundations of Database Technology", Springer-Verlag, 2000.
43. E. Teniente, "Analysis Pattern Definition in the UML", Proceedings Information Resources Management Association (IRMA) 2003, Idea Group Pub., pp. 774–777.
44. R.Wieringa, W. de Jorge, P.Spruit, "Using Dynamic Classes and Role Classes to Model Object Migration", Theory and Practice of Object Systems, 1(1), 1995,pp. 61-83.
45. R. K. Wong, H. L. Chau, F. H. Lochovsky, "A Data Model and Semantics of Objects with Dynamic Roles", 13th International Conference on Data Engineering, IEEE Computer Society, pp. 402-411.

Appendix A

```
public class Person
{
  public String name;
  public PhoneNumber phone;
  public Date birthDate;
  public Address address;
  Vector rols=new Vector()4;

  public double age()   { //Age calculation}
```

[4] Note that Person has a single multivalued attribute to store all the roles of that person, instead of having a different multivalued attribute for each of its roles (an attribute for the student instances, another for the employee instances…). We can use a single attribute since all the classes in Java are implicit subclasses of the class Object. When dealing with the attribute we make the appropriate castings to the specific role class.

```
  public void addRole(Object o) //Adding a new role
  {
    if (o instanceof Employee)
    { //Checking mayBeHired constraint
      int i=0; int numSt=0; Object o2;
      while (i<rols.size() && numSt<2)
      {
        o2=rols.get(i);
        if(o2 instanceof Student) numSt++;
        i++;
      }
      if(numSt<2) {rols.add(o);
           ((Employee)o).naturalEntityType=this;}
      else System.out.println("Error");
    }
    . . .
  }

  public void deleteRole(Object o)
  {
    if(o instanceof Employee)//Checking mayBeFired
                                           constraint
    {
     if(!((Employee) o).
                 workingStatus.equals("MaternityLeave"))
     { rols.removeElement(o);
           ((Employee) o).naturalEntityType=null;}
    }
    // ...
  }
}

public class Employee
{
  public int emp;
  public String category;
  public Object naturalEntityType;
     . . .
  //Adopted properties
  public String name()
     { return ((Person) naturalEntityType).name; }
  public double age()
     { return ((Person) naturalEntityType).age; }
}
```