Towards a More Suitable Class Hierarchy for Persistent Object Management

Adeline Capouillez, Robert Chignoli, Pierre Crescenzo, and Philippe Lahire[1]

1 Introduction

The aim of our study is to improve the description power of the class hierarchy. Thus, we want to facilitate the reusability of persistent objects in the context of several applications which share persistent data. To this end, we intend to specify more precisely the relationships between classes. The new information provides a more accurate and flexible class hierarchy. So we could load and update the persistent objects which have not got the exactly adequate structure for the transient schema. This can happen when applications evolve regardless of the persistent schema of classes.

The fact that several applications, at different evolution steps, access the same persistent objects implies two possibilities:

Partial schema of classes Some applications may only have a partial knowledge of the persistent schema of classes. The instances of known classes are of course directly accessible. However we may also want to load other persistent objects that *can be seen as* instances of known classes.

Evolution of classes The classes of an application can evolve. The persistent instances stored by the former versions of these classes should be able to be loaded, used, and even translated in order to be adapted to the new versions.

The second situation, the *evolution of classes*, will be dealt with with the management of specialized version relationships such as those of the *Presage* system [Tal94]. In this paper, we shall only present elements of a solution to the first situation.

In section 2 we will first present the context of our work, then we shall show the contributions of the relationship information thanks to the example of a generalization relationship in section 3. We may have chosen a specialization relationship but generalization is a little more original. For this example, we shall present the conditions needed to establish the relationship. Then we shall study the loading and updating phases and we shall detail the different resulting situations. For these situations, we shall give arising constraints and operations to perform. We want thus to demonstrate the interest of the relationships between classes associated to more accurate semantics in order to improve the quality of the schema of classes and to share persistent objects. We will conclude with an overview of possible future works.

[1]For all Authors: Laboratoire I3S (UNSA/CNRS), Team OCL, 2000 route des lucioles, Les Algorithmes bâtiment Euclide B, BP 121, F-06903 Sophia Antipolis CEDEX, France. E-Mails: {Adeline.Capouillez | Robert.Chignoli | Pierre.Crescenzo | Philippe.Lahire}@unice.fr

2 Framework of the study

To improve the class hierarchy power, we have defined a meta-object model called *OFL* [CCL99a, CCCL00a]. To facilitate your understanding of the rest of this paper, we need to present some elements of this model and the context of this study.

2.1 The *OFL* model

The *OFL* model, which is the basis of this work, is defined to bring out the notion of *relationship between classes* in the object-oriented languages (such as *Java* [GJS96], *Eiffel* [Mey92], or *C++* [Str97]). *OFL* is designed in the software engineering context [Ous99]. It describes for each language one language-concept entity which manages one or several description-concepts. These description-concepts represent the different *kinds* of classes (for example, in *Java*, we can find classes, interfaces, arrays, ...). Each of them can be considered as the *source* or as the *target* of a relationship (described by a relationship-concept) such as inheritance or aggregation.

Hereafter, we present the few elements of *OFL* that are essential for the understanding of this paper.

- The system is fully reified: the classes (such as in *CLOS* [Kee89] or *Smalltalk* [GR83]) and the relationships are also described as instances.
- The feature definition (functions, procedures and attributes) and the invariant (of class), described under the form of a conjunction of conditions, are stored within classes.
- The values of the attributes are stored within instances.
- When we speak about the *type of a feature*, we mean:
  - for an attribute: its type,
  - for a procedure: the set of types of its parameters,
  - for a function: the set of types of its returned result and its parameters (the returned result is considered as a result-parameter which only provides a syntactic simplification).
- Each class defines a default value for each of its attributes. This default value must respect the invariant of the class.

The main original aspect of our approach is to focus on the properties of the relationship-concepts (relationships between classes) in order to exploit these data. The first interest of this rich description is that we can use this new information to improve the quality of the developed software. Therefore we can provide better documentation, maintainability, reusability, ... Another interest is to be able to make a better specification of the relationships between classes in object-oriented languages. For example, we can set a real specialization or generalization (or ...) relationship, as in the modelling stage (*UML* [RJB98]), between two classes rather than just using inheritance as a roundabout way.

Unlike *Java*, *C++*, *Eiffel*, ..., each of which offers an inheritance relationship with fixed semantics, our approach is to propose a more flexible way to design more adequate relationships. Like *CLOS* and *Smalltalk*, we can redefine the operational semantics of inheritance or even define new relationships. But unlike them, we want to offer the programmer a simple way to do that [CCL99b].

This paper neither presents the *OFL* model nor the way to construct new relationships. We only want to show here some possible improvement, that we may get for the management of persistent object thanks to a better classification.

2.2 Context

We are first in the context of a persistent programming language which does not rely on a database management system. So some problems may appear. For example, when you load an object, in an object-oriented database management system, you automatically load its class. We assume a persistent programming language which would not proceed this way. Indeed, as said in the introduction, an application may have evolved regardless of the persistent schema, but we think that we can even so provide the loading of the object.

Thus we want to point out that we are in the framework of a programming language where the loading of a class from the persistent schema is not performed implicitly[2]. Therefore, loading an instance does not imply loading its class. Our approach is indeed to load this instance by *adapting* it to the transient schema (the application one). We admit that it is also possible to load a flattened view[3] of the class. We do not assume that the loading operation is more or less static or dynamic.

We have chosen to use the *ROOPS* service [Cap99] which provides a persistent modelling of *OFL* entities. *ROOPS* is designed in order to allow the storage of both instances and classes but also of all the information dealing with the relationships between classes[4].

To explain our approach, we shall now give a definition of the following terms: *migration*, *loading* and *updating*.

What is meant by migration is the process which allows to change the class of an object. It is not polymorphism which allows to consider an object as an instance of a compatible class. It is an irreversible transformation (unless we make an opposite migration which is not a cancellation but another transformation which cannot guarantee that the object will come back to its original state). Therefore the migration allows to break the instantiation relationship which exists between an instance and its class.

Loading is the operation which makes an object go from the persistent world to the transient one. The updating process is the reverse operation.

In the framework of our approach, we did not allow to perform the following operations during the updating process:

[2]The explicit loading of classes is obviously feasible.

[3]For a class, *flattened* means a transitive closure is made on this class. So all its features are seen as local.

[4]The relationships between classes and objects, such as that of the instantiation one, or between objects are also designed in *OFL* and *ROOPS*. But this paper does not intend to deal with these kinds of relationships.

- **Migration.** We consider that the change of class for an object is too important an operation and it cannot be made implicitly by an application when updating. Indeed, an application could *lose the track* of an instance that it created if another application makes this instance migrate.
- **Modifying the value of persistent attributes which are not loaded in the transient world.** Those attributes, which are not loaded by the application, must not be modified, in order to keep the integrity of persistent instances at the updating time.
- **Representating an object of the real world by several persistent instances.** In order to keep the integrity of the persistent world (each persistent object has a unique identity) at the updating time, if the transient image of the persistent instance is incompatible with this persistent instance, the creation of a new persistent instance corresponding to the same object is prohibited.

2.3 Caption

Finally, figure 1 gives the common caption of all the other figures of this document, therefore they will only show the specific part of their caption.

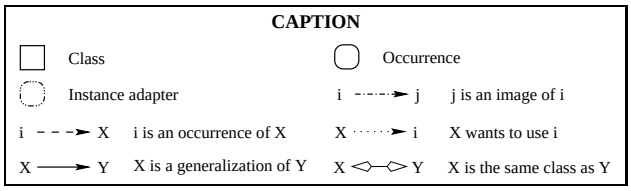

Figure 1: A common caption

j is an image of i means that j describes the same object as i but with another type. *X is the same class as Y* means that X, from the persistent world, is faithfully represented by Y in the transient world.

3 Generalization relationship

We choose to present a generalization relationship rather than a specialization one [CCCL00b] because it is more relevant to improve a class power hierarchy. This kind of link is useful in some situation. For example, we want to add a class in the middle of a library class hierarchy. And we do not want or cannot modify this library by *re-engineering* it with a top-down approach. Here, it is possible to use a generalization relationship to insert the new class with accurate links into the existing ones.

3.1 Definition of the relationship

A generalization relationship is the reverse of a specialization relationship. For lack of anything better, the inheritance implemented in the object-oriented languages

is sometimes used to implement generalization [Mey97]. In order to be able to establish a generalization relationship between a `S` source-class and a `C` target-class, it is necessary to satisfy the following conditions:

1. `S` cannot define new features.
2. `S` can remove some features from `C`.
3. `S` can redefine the features of `C` if and only if the type of redefined attributes, redefined feature parameters and redefined function results are generalized according to the type defined in `C`.
4. The invariant of `S` is equivalent to or less strict than the `C` one.
5. The set of the instances (extension) of `S` includes all the instances of `C`.

The three following examples present typical cases of generalization relationships:

1. The `RECTANGLE` and `LOZENGE` classes (source-classes) are generalizations of the `SQUARE` class (target-class).
2. The `CAR` class is a generalization of the `PORSCHE` class.
3. The `AIRCRAFT` class is a generalization of both `HELICOPTER` and `PLANE` classes.

3.2 Illustration: influences and contributions

To illustrate the influence of the generalization relationship in the management of persistent objects, we give the example described in figure 2. In the persistent world, the `CAR` class (which has a direct `a1` instance) is a generalization of the `DIESEL_CAR` class.

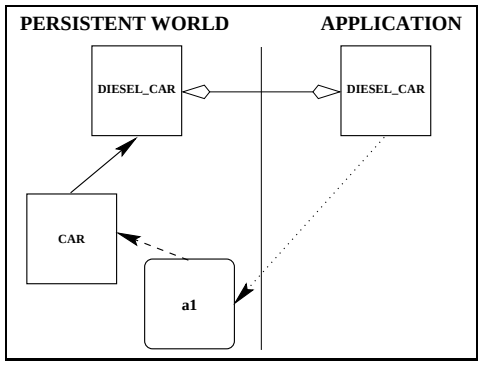

Figure 2: A generalization relationship

Here are some elements of the two class definitions (considering that `DIESEL_OIL` is a specialization of `FUEL`). It is not a source code but rather a flattened description of these classes.

```
Class CAR
    Features
        owner:  PERSON
        fuel:  FUEL
        consumption:  INTEGER
    Invariant
        consumption ≥ 0
End_Class CAR
```

```
Class DIESEL_CAR
    Features
        owner:  PERSON
        fuel:  DIESEL_OIL
        consumption:  INTEGER
        preheating_time:  INTEGER
    Invariant
        (consumption > 0) ∧
        (preheating_time ≥ 0)
End_Class DIESEL_CAR
```

The DIESEL_CAR class is loaded by an application A from the transient world. This class is stemming from the persistent world which also contains the CAR class. A has no knowledge of CAR. There is a persistent a1 instance of CAR. We can admit that the application A wants to handle all the persistent instances of DIESEL_CAR but also those of CAR which are compatible with the description of a DIESEL_CAR.

3.2.1 Loading

We can see that the DIESEL_CAR class has no instance, the CAR class has one. However, this instance can be viewed under some conditions as a DIESEL_CAR.

An a1 instance of CAR in the persistent world can become a d1 instance of DIESEL_CAR in the transient world, following the next chronological steps:

1. a1 is loaded in transient memory (let us call it a1-aux).
2. Each missing attribute from a1-aux according to DIESEL_CAR is added to a1-aux with its default value defined in DIESEL_CAR.
3. If and only if a1-aux satisfies the invariant of DIESEL_CAR, then it is viewed in the transient world as an instance of DIESEL_CAR called d1 (cf. figure 3).

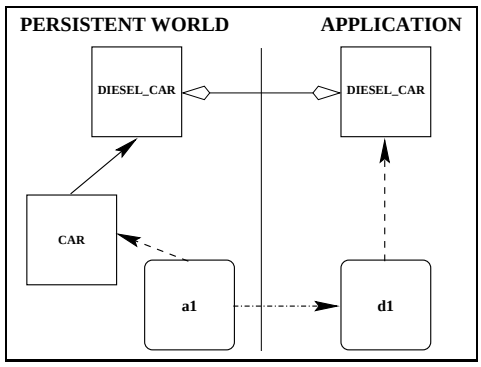

Figure 3: Loading of a generalized object

If the condition mentioned in the last step is not satisfied then a1-aux is removed from the transient world. Therefore loading a1 is impossible.

During the adaptation from a1 to d1, we neither deal with the invariants nor the routines because they are described at the class level and not at the instance level.

3.2.2 Updating

When all the operations are finished in the transient world, we deal with the updating phase in the persistent world. Several situations can occur:

No updating is wanted. All the modifications made in the transient world are lost.

An updating is wanted. Here we face two alternatives:

- **No value of an attribute added to `d1` has been modified**[5]**.** In this case, it is useless to keep the value of these attributes. `a1` from the persistent world is therefore updated according to the attributes of `d1` defined in `CAR` (cf. figure 4). Moreover this is directly possible because the invariant of `DIESEL_CAR` is compatible with the `CAR` invariant. Indeed, this compatibility is ensured by the semantics of the generalization relationship.

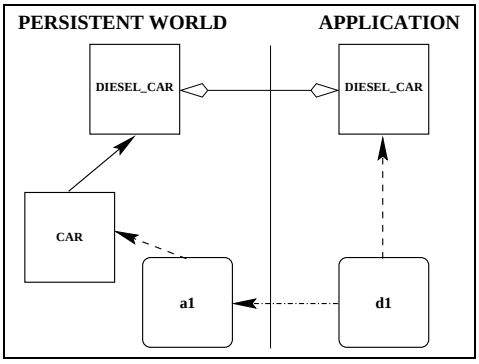

Figure 4: Updating of a generalized object (particular case)

- **The value of at least one attribute added to `d1` has been modified.** We want to keep `a1` from the persistent world as a direct instance of `CAR`. We also want to keep the new information brought by `A` which considers `a1` as a `DIESEL_CAR`. To this purpose, we add an *adapter* to `a1` in the persistent world. It allows to consider `a1` as a direct instance of `DIESEL_CAR`. This adapter called `d1-a1` contains all the values of the direct attributes of `DIESEL_CAR`. In our example, we keep all the values of the attributes of `d1` that are not in `a1`[6] (cf. figure 5). The values of the attributes of `d1` contained by `CAR` are updated in `a1`, those specific to `DIESEL_CAR` are updated in `d1-a1`. An adapter can be the interface of only one instance. An instance can have several adapters, each of them being attached to a different type[7].

[5] They still have their default value.
[6] Hence the notation `d1-a1`: `d1` minus `a1`.
[7] It means an object can have several instantiation relationships to different classes.

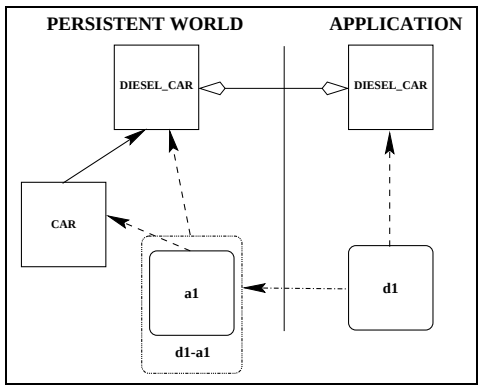

Figure 5: Updating of a generalized object

4 Prospects and conclusion

Thanks to the studied example, this paper has presented our first works on the use of information associated to the relationship between classes in order to improve the class hierarchy quality and to reuse persistent objects.

In this example, the use of a specific relationship (generalization) shows that it is more pertinent than a simple inheritance relationship. Indeed, inheritance can be used for numerous uses (such as specialization, generalization, views, versions, code reuse, ...). It is therefore impossible for the system to attach some strong semantics to the edges (inheritance relationship) of the schema of classes. It is even more difficult to use these semantics when the instances are loaded by applications which only know a part of this schema. We have not shown it here, but the use of a specialization relationship improves the persistent object reuse too.

We have also shown that a better knowledge of the relationships between classes — at the persistent level as well as in transient applications — allows to handle instances which, otherwise, would not be *loadable* by applications.

These are our development prospects:

- the generalization of this approach to version relationships to handle the application evolution,
- an extension of this approach removing some of the constraints set in the context section (for example, we could accept migration in some situations in order to re-classify objects), and
- the programming of a prototype handling a subset of the *OFL* model, for example by extending *Java* with one or several new relationships.

References

[Cap99] A. Capouillez. ROOPS : un service paramétrable de persistance pour OFL. Technical Report 99-15, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, septembre 1999.

[CCCL00a] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Gestion des objets persistants grâce aux liens entre classes. In *Conférence OCM'2000 (Objets, Composants, Modèles 2000)*, mai 2000.

[CCCL00b] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. How to Improve Persistent Object Management using Relationship Information? (to appear). In *Conference WOON'2000 (4^{th} International Conference "The White Object Oriented Nights" 2000)*, June 2000.

[CCL99a] R. Chignoli, P. Crescenzo, and P. Lahire. An Open Object Model based on Class and Link Semantics Customization. Technical Report 99-08, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, March 1999.

[CCL99b] R. Chignoli, P. Crescenzo, and P. Lahire. Customization of Links between Classes. Technical Report 99-18, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, November 1999.

[GJS96] J. Gosling, B Joy, and G Steele. *The Java Language Specification*. The Sun Microsystems Press Java Series. Sun Microsystems, 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 — The Language and its Implementation*. Computer Science. Addison-Wesley Publishing Co., 1983.

[Kee89] S. Keene. *Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS*. Addison-Wesley Publishing Co., 1989.

[Mey92] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 1997.

[Ous99] C. Oussalah, editor. *Génie objet : analyse et conception de l'évolution*. Hermes, septembre 1999.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. The Object Technology Series. Addison-Wesley Publishing Co., December 1998.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., 3 edition, 1997.

[Tal94] G. Talens. *Gestion des objets simples et composites*. Thèse de Doctorat en Génie Informatique, Automatique et Traitement du Signal, Université Montpellier II, février 1994.