# How to Improve Persistent-Object Management using Relationship Information?

Adeline Capouillez, Robert Chignoli, Pierre Crescenzo, Philippe Lahire

# How to Improve Persistent-Object Management using Relationship Information?

*May 23, 2000*

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

**Abstract:** The *OFL* model proposes a reified description of the main concepts included in the object-oriented languages based on classes. With this model and one of its interesting characteristics — the ability to create and/or parameterize the relationships between classes such as inheritance — we aim to show that we can benefit from the information associated with these relationships when they are used in the framework of applications which share persistent data. Therefore we shall develop examples to show this contribution through two relationships: specialization and generalization of class. For each of these examples, we 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 will give arising constraints and operations to perform. Thus, we want to demonstrate the interest of such relationships between classes associated with more accurate semantics to share persistent objects.

**Keywords:** Persistence, Relationship, Class, Specialization, Generalization, Evolution

## 1 Introduction

The aim of our study is to improve the sharing of persistent objects between different applications. To this end we want to decrease the dependence of those objects with the structure of the schema of classes. The means that we use to achieve this objective is to take advantage of the semantic information provided by the import relationships between classes. Indeed, this information allows us to loosen instantiation relationships, without breaking them, while an application is running.

The fact that several applications access the same persistent objects implies two possibilities:

**Partial schema of classes** Some applications may have only a partial knowledge of the persistent schema of classes. The instances of known classes are of course

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

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 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 for the first situation.

In section 2 we will first present the context of our work, then we will show the contributions of the relationship information thanks to the two following examples: specialization relationship in section 3 and generalization relationship in section 4. For each of these examples, we 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 will give arising constraints and operations to perform. We want thus to demonstrate the interest of relationships between classes associated to more accurate semantics to share persistent objects. We will conclude with an overview of possible future works.

# 2 Framework of the study

## 2.1 *OFL* model

The *OFL* model [CCL99a, CCCL00], 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, AG98, Fla99], *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* which are mandatory for the understanding of this paper.

- The system is fully reified: the classes (such as in *CLOS* or *Smalltalk*) 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 conjunction of conditions, are stored within classes.
- The values of the attributes are stored within instances.
- When we speak about *type of a feature*, this means:
  - for an attribute: its type,

  - for a procedure: the set of types of its parameters,
  - for a function: the set of types of its return and its parameters (the return is considered as a result-parameter which provides only a syntactic simplification).

- Each class defines a default value for each of its attributes.

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, maintenability, 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 modeling stage (*UML* [BJR98, RJB98, JBR99]), between two classes rather than using inheritance as a roundabout way.

Unlike *Java*, *C++*, *Eiffel*, ..., each of which offers an inheritance relationship with fixed semantics, we want to propose a more flexible way to design more adequate relationships. Like *CLOS* [Kee89] and *Smalltalk* [GR83], 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 does not present the *OFL* model nor the way to construct new relationships. We only want to show here some improvements to object-oriented programming within the framework of persistence.

## 2.2 Context

First, we are in the context of a persistent programming language which does not rely on a database management system. So some problems may appear. For example, in an object-oriented database management system, when you load an object, 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 independently of the persistent schema, but we think that we can even so provide 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 make any assumption on the fact that the loading operation is more or less static or dynamic.

We have chosen to use the *ROOPS* service [Cap99] which provides a persistent modeling of *OFL* entities. *ROOPS* is designed in order to allow the storage of both

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

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

instances and classes but also of all the information dealing with the relationships between classes[4]. The aim is to control and maintain, the persistent information, with as much accuracy as possible. Therefore, the persistent representation of classes and relationships are here implemented in order to improve the use of persistent instances[5] but not to allow the dynamic loading of classes and relationships. This is obviously feasible in another context.

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 the polymorphism which allows to consider an object as an instance of a compatible class. It is an irreversible transformation (unless we do 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.

The 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:

- **The migration.** We consider that the change of class for an object is too much important an operation and it can not 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.
- **The modification of the value of persistent attributes which are not loaded in the transient world.** In order to keep the integrity of persistent instances at the updating time, those attributes, which are not loaded by the application, must not be modified.
- **The representation of an object of the real world by several persistent instances.** In order to keep the integrity of the persistent world (any 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.

*`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.

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

[5]An improvement can actually be expected because the structural information is more precise.

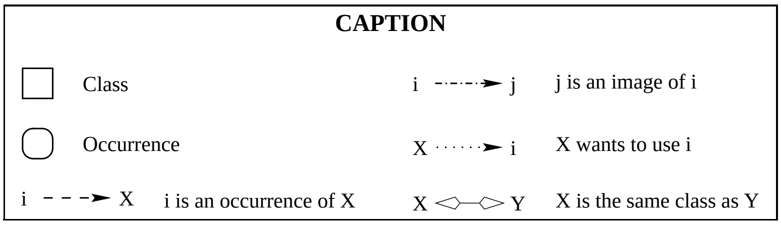


Figure 1: A common caption

# 3 Specialization relationship

## 3.1 Definition of the relationship

A specialization relationship defines a relationship between a source-class and a target-class. Inheritance, which is generally present in the object-oriented languages and which can also be found in *UML*, is a good approximation of this relationship [Mey97]. The necessary and sufficient conditions to be able to establish a specialization relationship between the `S` source-class and the `C` target-class are:

1. `S` owns all the features of `C`.
2. `S` can add new features to `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 specialized according to the type defined in `C` (covariance).
4. The invariant of `S` satisfies the invariant of `C`.
5. All the instances of `S` are also instances of `C`.

The two following examples present typical cases of specialization:

1. The `SQUARE` class (source-class) is a specialization of the `RECTANGLE` and `LOZENGE` classes (target-classes).
2. The `PORSCHE` class is a specialization of the `CAR` class.

## 3.2 Illustration: influences and contributions

To illustrate the use of the knowledge of a specialization relationship for the management of persistence, we give the example described in figure 2. In the persistent world, the `DIESEL_CAR` class (which has a direct `d1` instance) is a specialization of the `CAR` class.

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

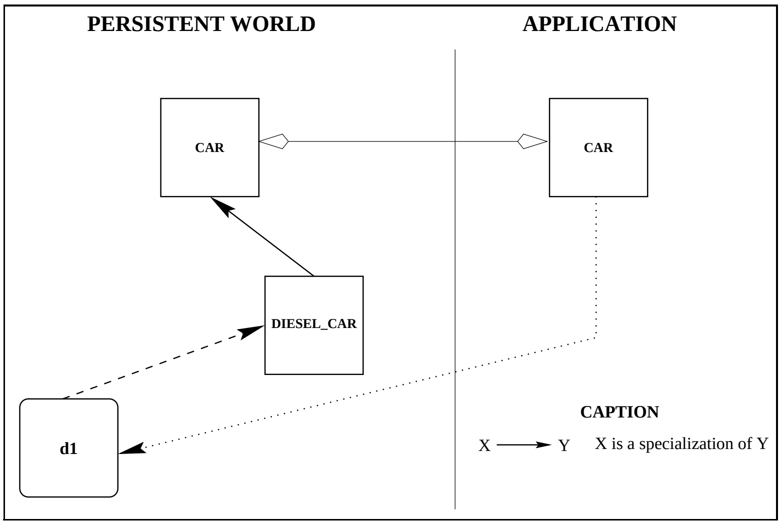


Figure 2: A specialization relationship

```
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
```

In the transient world, an application `A` loads the `CAR` class from the persistent world. But, `A` does not know the existence of the `DIESEL_CAR` class (which has been created by or for other applications). `A` uses the instances of `CAR` of the persistent world and/or creates new ones. In order to illustrate the use of a specialization relationship, we focus on an example: in `A`, we want to handle all the instances of `CAR`[6].

### 3.2.1 Loading

Thanks to our figure we can see that `CAR` has not got any direct instance. However, the specialization relationship which joins `CAR` to `DIESEL_CAR` allows the polymorphism (as inheritance). A consequence is that all the instances of `DIESEL_CAR` are also instances of `CAR`. Indeed, we have to handle, in the transient world, the object `d1` as a `CAR`[7] but not as a `DIESEL_CAR`.

It is obvious that the loading of `d1` cannot be made directly. We must adapt it to its new definition, that of the `CAR` class (cf. figure 3). For this reason and because it is a specialization relationship, it is necessary to perform the following

[6]The access to instances is provided either by the application or the persistent world.

[7]`CAR` must be concrete, so all its features are fully implemented.

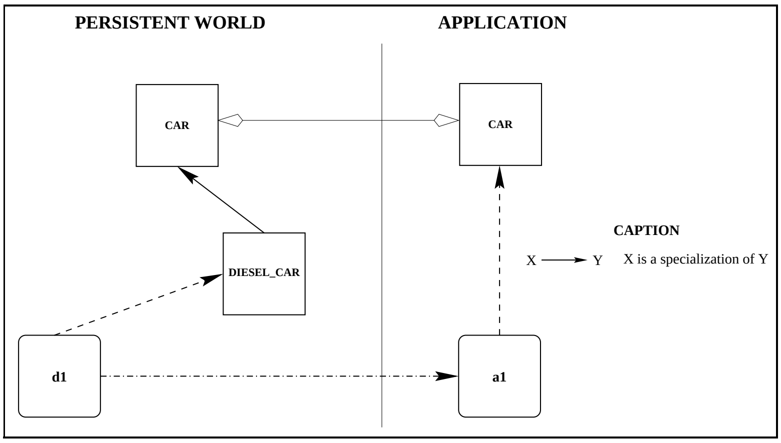


Figure 3: Loading of a specialized object

operation to switch from the persistent instance to its transient image called `a1`. We must remove the value of the attributes added by `DIESEL_CAR` to `CAR` (value of `preheating_time`). As we have seen in the context, attributes and methods are stored in the classes. Therefore, it is not useful to care about them at the instance level.

Likewise, invariants are stored within classes while instances store only attribute values. Therefore, the object `d1` does not describe the type of its features: the `DIESEL_CAR` class does it. If some of these features have been redefined according to `CAR`, then they inevitably have been specialized. So their value in `d1` remains of course compatible. As a consequence, no particular adaptation is necessary on the type of features.

### 3.2.2 Updating

When the application `A` has used (and modified or not) `a1`, the user is faced with two situations while updating the persistent world according to the state of the transient world:

**No updating is wanted.** Application `A` uses persistent data but does not want to propagate any of its modifications to the persistent world. Thus, there is nothing to do in such a case.

**An updating is wanted.** This is possible only if the value of each attribute (of `a1`) specialized in `DIESEL_CAR` is compatible with its type in `DIESEL_CAR`. It is also necessary that `a1`, to which the direct attributes of `DIESEL_CAR` had been added with their value from `d1`, satisfies the invariant of `DIESEL_CAR` (cf. figure 4).

If these two conditions are not satisfied, `A` is notified that the updating (understood as: without making `d1` migrate, without modifying the value of its attributes that were not handled by `A` and without creating a new persistent object which is not dependent on `d1`) is impossible. It is not possible because we cannot make `d1`

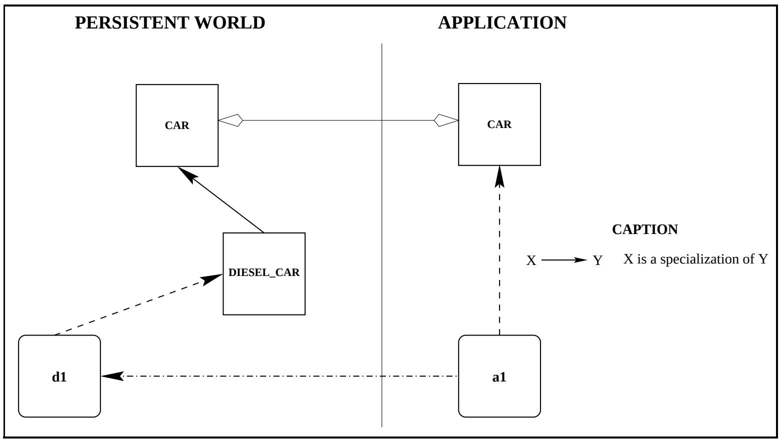


Figure 4: Updating of a specialized object

migrate. To modify the value of the attributes that were not used by A and to create a new persistent object which is independent from d1 are also forbidden.

It is obvious that any direct instance of a class[8] in the persistent world can always be loaded and updated without any difficulty. The problem happens only for indirect instances.

#### 3.2.3 Another situation

We can study the reverse situation. In the persistent world of figure 5, there is an a2 instance of the CAR class (of which DIESEL_CAR is a specialization). The application B of the transient world loads the DIESEL_CAR class but not CAR[9]. In B, we want to handle all the persistent instances of DIESEL_CAR as well as all the instances of CAR which are compatible with DIESEL_CAR (*i. e.* "all the CARs that could be DIESEL_CARs" or else "all the direct instances of CAR which satisfy the conditions of a DIESEL_CAR").

This problem is solved, in section 4, bearing in mind that specialization is the reverse of generalization.

# 4 Generalization relationship

## 4.1 Definition of the relationship

A generalization relationship is the reverse of a specialization relationship described in section 3. For lack of anything better, the inheritance present in the object-oriented languages is sometimes used to implement a generalization [Mey97]. In

[8]In our version relationships, which will be presented in a future paper, each version of a class would be itself a class with its direct instances.

[9]Two situations can occur. Either CAR has been added to the persistent schema of classes after the design of B, or the designer of B has loaded a flattened view of DIESEL_CAR without taking care of the remaining part of the persistent schema of classes.

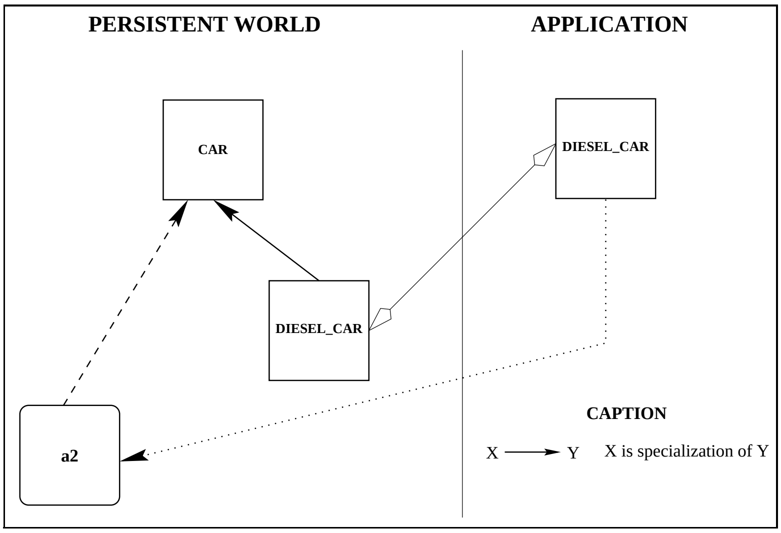


Figure 5: Another configuration for a specialization relationship

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 or less strict than the C one.
5. The set of the instances (extension) of S includes all the instances of C.

The two examples of the specialization relationship can be analysed again for the generalization relationship and we add a new one:

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.

## 4.2 Illustration: influences and contributions

To illustrate the influence of the generalization relationship in the management of persistent objects, we reuse the CAR and DIESEL_CAR classes defined in section 3.2. As in generalization, this example is presented in figure 6.

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

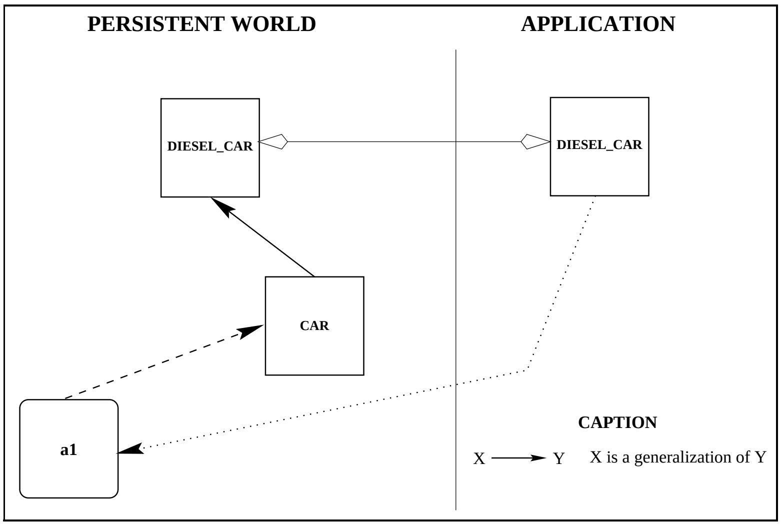


Figure 6: A generalization relationship

(a generalization of `DIESEL_CAR`). `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`.

### 4.2.1 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 in the transient world, it is viewed as an instance of `DIESEL_CAR` called `d1` (cf. figure 7).

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

As for the specialization (cf. section 3.2.1) during the adaptation from `a1` to `d1`, we do not address neither the invariants nor the routines because they are described at the class level and not at the instance level.

### 4.2.2 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:

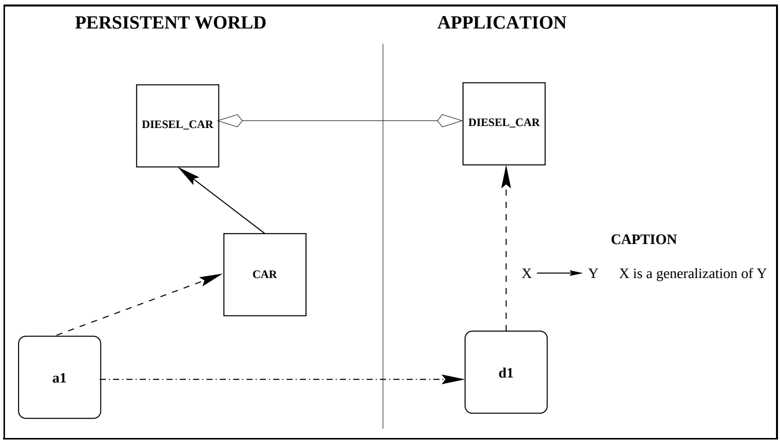


Figure 7: Loading of a generalized object

**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` have been modified[10].** 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 8). 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.

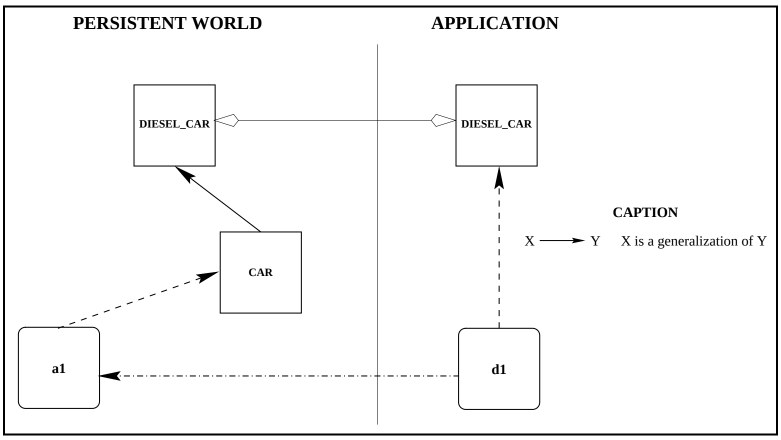


Figure 8: Updating of a generalized object (particular case)

- **The value of at least one attribute added to `d1` have 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

[10] They still have their default value.

DIESEL_CAR. In 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[11] (cf. figure 9). 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[12].

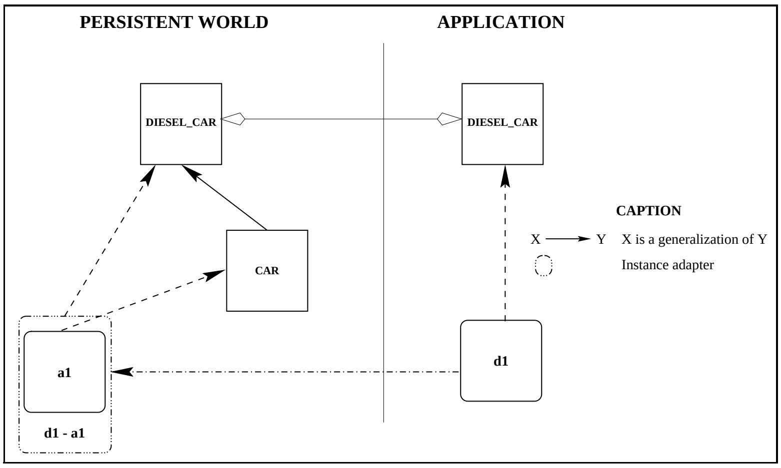


Figure 9: Updating of a generalized object

### 4.2.3 Another situation

We will study the reverse situation. In the persistent world, we find the CAR class which generalizes DIESEL_CAR (with a d2 instance). In the transient world, an application B only loads CAR. B wants to handle all the CARs of the persistent world. It is easy to notice on figure 10 that it is the same configuration that of specialization described in section 3.

# 5 Prospects and conclusion

Thanks to the two examples studied, this paper has presented our first works on the use of information associated to the relationship between classes in order to manage persistent objects.

In these examples, the use of specific relationships (specialization and generalization) shows that they are 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

[11] Hence the notation d1-a1: d1 minus a1.

[12] It means an object can have several instantiation relationships to different classes.

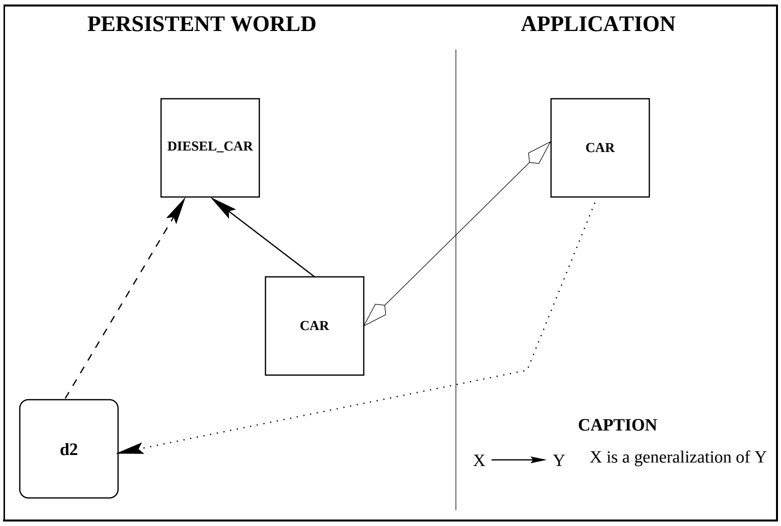


Figure 10: Another configuration for a generalization relationship

of classes. It is even more difficult to use this semantics when the instances are loaded by applications which only know a part of this schema.

We have also shown that a better knowledge of 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,
- the study of the influence of use relationships (such as aggregation or composition) in addition to that of the import relationships (of which inheritance is the spearhead) which has been made in this paper,
- an extension of this approach removing some of the constraints set in the context section (for example, we could accept migration in some situations), 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

[AG98] K. Arnold and J. Gosling. *The Java Programming Language.* The Java Serie... from the Source. Sun Microsystems, 2 edition, 1998.

[BJR98] G Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide.* The Object Technology Series. Addison-Wesley Publishing Co., October 1998.

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

[CCCL00] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Gestion des objets persistants grâce aux liens entre classes (à paraître). In *Conférence* Objets, Composants, Modèles *2000*, mai 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 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 Informatique, Signaux et Systèmes de Sophia-Antipolis, November 1999.

[Fla99] D. Flanagan. *Java in a Nutshell: a Desktop Quick Reference.* O'Reilly, 3 edition, December 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.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Software Development Process.* The Object Technology Series. Addison-Wesley Publishing Co., January 1999.

[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, 2 edition, 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.