Design Patterns for Persistence Management in Reflective Object-Oriented Languages

Stéphane Demphlous **Franck Lebastard**

CERMICS / INRIA Database Team
2004, route des Lucioles,
B.P.93, F-06902 Sophia Antipolis Cedex, France
Email: {Stephane.Demphlous, Franck.Lebastard}@sophia.inria.fr

ABSTRACT

The reflection paradigm has proved to be useful both for both programming languages and databases. We think that there will be a growing need to design, with an object-oriented reflective language, persistent applications that are client of a server database management system. These applications can themselves include several instanciation levels. In this research report we consider the ways to manage persistence in reflective object-oriented programming languages. We justify why persistence can be added through a class library. Then we propose several design patterns, depending on the chosen persistence paradigm, the constraints enforced by the language, or those enforced by the designer. These design patterns take into account some important properties of most of reflective languages, in particular, the ability to define several instanciation levels. So the designer of an application can choose one of the proposed object models, according to her/his needs and the properties of her/his language.

RESUME

Le paradigme de la réflexion a témoigné de son utilité à la fois dans le domaine des langages de programmation et dans celui des bases de données. Nous pensons que l'avenir apportera des besoins croissants de définition de clients de systèmes de gestion de bases de données serveur au moyen de langages de programmation réflexifs à objets. Ces applications peuvent elles-mêmes comporter plusieurs niveaux d'instanciation. Dans ce rapport de recherche, nous prenons en considération les moyens de gérer la persistance dans des langages de programmation réflexifs à objets. Nous justifions les raisons pour lesquelles la persistance peut être ajoutée au moyen d'une bibliothèque de classes. Puis nous proposons plusieurs patrons de conception, compte tenu du paradigme de persistance choisi, des contraintes imposées par le langage, ou de celles imposées par le concepteur. Ces patrons de conception prennent en considération des propriétés importantes communes à la plupart des langages réflexifs, et en particulier la possibilité de définir plusieurs niveaux d'instanciation. Ainsi, le concepteur d'une application peut choisir un des modèles à objets proposés en fonction de ses besoins et des propriétés du langage.

1. INTRODUCTION

A key to design modern applications is to use the client-server paradigm. When an application needs to create or to manipulate persistent data, a good policy is to design a persistent client connected to a server database management system (DBMS).

Now, the reflection, defined in [2] as the ability for a program to manipulate as data something representing its state during its own execution, and the reflection paradigm [5, 10, 19] have become a major issue for system definition. Languages and tools [1, 11, 12] have widened the use of reflection. So client persistent applications may be built using a reflective programming language, or can be themselves reflective.

Moreover, adding reflective properties to DBMSs appears to be very promising: it helps to adapt the DBMS data model, that is, the formalism in which user data and their definition (metadata) are expressed and controlled. The intercession property of many reflective systems, defined in [2] as the ability for a program to modify its own execution state or alter its own interpretation or meaning, can be applied to a DBMS, allowing the user to modify or to extend the data model. Several studies have shown the interest of this approach [9, 16, 17]. Vodak [17] and Tigukat [24] are fully reflective object-oriented DBMSs (OODBMS).

So we believe that, in a near future, there will be growing needs of persistence management in reflective models with intercession properties.

One of the interests of a reflective object-oriented system is to be described as an object model. The persistence properties of such a system should be described as a part of this object model. An aim of this research report is to show how a persistent reflective object model can be defined. But we widen the scope. Some reflective object-oriented languages are actually used by designers – for example, Smalltalk [12] or CLOS [1]. These designers would probably like to design persistent applications in these transient reflective languages, for example, in order to encapsulate an existing, non reflective DBMS by an end-user application having introspection properties. In that case, we would not provide for the needs of the designers by showing that a persistent reflective language can be created. As a matter of fact, we would implicitly ask the designer to modify her/his language, or to rewrite it from scratch. A more useful work would be to show how persistent applications can be designed in transient reflective languages.

So there is a need to offer design patterns for persistence libraries in object-oriented reflective languages. These languages have some specific properties. There are objects, called metaobjects, that represent elements of the programs or of the language itself. For example, the classes are objects, called class metaobjects, instance of metaclass metaobjects [15]. The provided design patterns

- should be language independent;
- should take into account some properties specific to reflection, like the instanciation relationships between objects, or the different instanciation levels;
- and, finally, should solve the problems induced by the coexistence of transient and persistent objects or metaobjects in a same application.

In this paper, we aim at providing such design patterns, and justifying them. This research report is divided in six parts. We first show in the section 2 why persistence abilities can be, and should be, added to a class-based object-oriented reflective system using an object model. We show that the management of an instanciation branch is one of the specific problems encountered when designing a class library in such a reflective language. We propose then, in the section 3, several object models to manage persistence, according to the capabilities of the language. These object models raise some problems related to the management of transient and persistent classes, and to the compatibility of metaclasses. They are treated in the sections 4 and 5. Then, in the section 6, we apply our proposals to the reflective object-oriented language Power Classes. Finally, we consider the related works and we conclude the research report.

2. OBJECT PERSISTENCE IN A REFLECTIVE CLIENT

2.1. The persistence property: a consequence of message passing

In many persistent languages, the persistent objects are instances of a common class because they must receive messages related to persistence, like locking messages [4, 20]. In short, the persistent objects must be instances of a common class in order to be manipulated as persistent objects. However, the possible need of an object to be instance of a specific class, or to share a specific metaclass, in order to be persistent has not been clearly shown in previous works. We think this is an important question since the answer may justify our will to design a class library in order to add persistence to a system. We study this problem in the following lines and we show that the persistence property of an object can be managed only by message passing.

Creating proxies

Let us assume for the moment that, when created, a persistent object is, in the client, a proxy of the actual object in the server OODBMS. When, in the client, an instanciation message is sent to a class *c*, a same message is sent, in the server, to the class *(c)*$_{server}$ associated to *c*. An object is created in the server and a proxy, associated to this newly created object, is created in the client. This architecture implies that a special, that is, persistent, instanciation message is sent to the class *c*. In order to receive a persistent instanciation message, a class must be instance of a metaclass on which this behavior has been defined. So we need to define a special, devoted to persistence, metaclass.

In this approach, the persistence is defined at the meta level. A class does not need to inherit from a special class in order to allow its own instances to be persistent, but must be itself instance of a special metaclass.

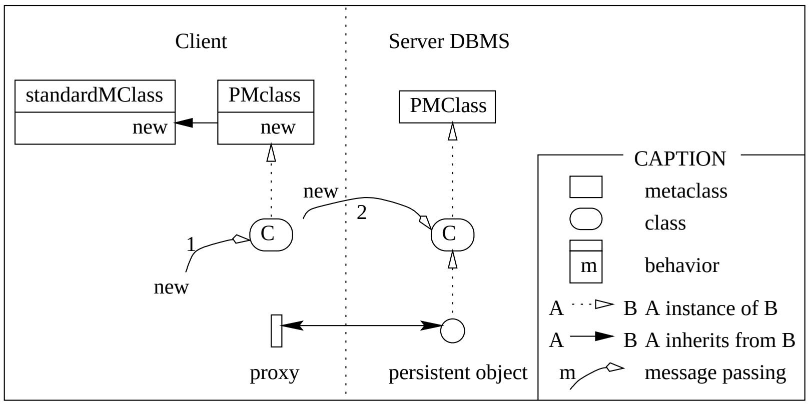

Figure 1. Instanciating objects on the server and defining proxies in the client.

This approach is summarized in the figure 1.

Replicating persistent objects in the server DBMS

A different point of view can be taken. One can consider that an object is first transiently created in the client, and then it can be replicated in the server OODBMS. A transient object becomes persistent when it is replicated. This architecture can be interesting: it can be useful to keep two versions of a persistent object, one in the client, and one in the server, to minimize the communications between the client and the server.

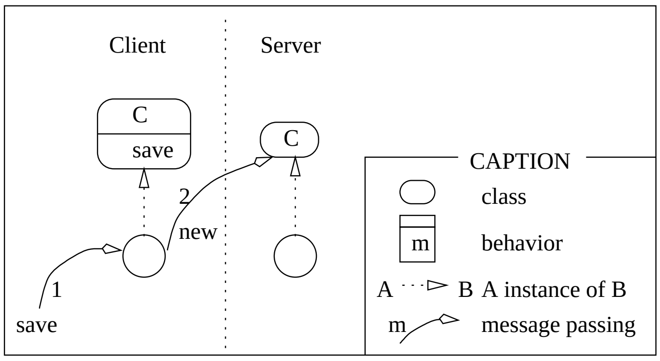

Figure 2. Replicating transient object.

We think that the object replication must be realized by message passing. First, replicating an object *o* on a server means, semantically, performing an operation on *o*. Moreover this replication can lead to modifications of *o* itself, for example it may store any kind of reference to the replicated object in the server. According to the object-oriented paradigm, the operations on objects have to be realized by message passing.

This second approach is summarized in the figure 2. A message *save* is sent to an object *o* instance of a class *c*. The class $(c)_{server}$ associated, in the server, to the class *c*, is then instanciated.

Persistence and message passing

So we state that, in a client-server environment, when the client uses an object-oriented reflective model, the creation of persistent objects is always the consequence of message passing.

- □ The use of proxies implies that persistent objects share a common metaclass;

- □ The use of the replication approach implies that the persistent objects be instances of a common class.

Now we have to decide whether these persistence-related behaviors should be closely integrated to the language or should be designed in an additional class library.

2.2. Persistent languages and persistent libraries

Two approaches can be taken to manage persistence in a reflective object-oriented language.

The first one is to design the native metaobject protocol of the language so that the whole language has persistent abilities.

Let us consider the ObjVLisp model [5]. In this model, the class `object` is the root of every inheritance tree. The class `class` is a metaclass that inherits from `object`. The class `class` is instance of itself and a class is instance of `class`.

- ☐ If we want to use the proxy approach, then we can define a new instanciation behavior on the class `class`, that creates persistent objects on the server.

- ☐ But if we want to use the replication approach, we have to define, on the class `object`, a behavior that replicates the target object on the server.

Since every object in this language is instance of a class that inherits from the class `object` and is itself an instance either of the class `class` itself, or of a subclass of `class`, any application will be persistent.

This example shows that a persistent and reflective object-oriented language can be built.

However, in this paper, we have a different point of view. We consider that a designer may want to build, with a programming language that is not natively persistent, an application managing persistent objects. In this case, the following question arises: how can we design, whatever the transient language may be, a library that manages persistent objects, metaobjects, classes and metaclasses?

2.3. Management of an instanciation branch

We show now that the previous questions are related to a problem that is specific to class-based reflective languages: the management of different levels of instanciation relationships.

We first define the following notations that we will use in the rest of this paper.

Notations

Let us assume that an object o is a direct instance of a class c, that is, c is the most specialized class of o. We use the notation $o \lhd c$.

Now, let us consider an object o_i, with $i \geq 0$. We state that o_{i+k} is the most specialized meta^{k-1}class of o_i. That is, o_i is a direct instance of o_{i+1} ($o_i \lhd o_{i+1}$), o_{i+1} is itself a direct instance of o_{i+2} ($o_{i+1} \lhd o_{i+2}$) and so on, till o_{i+k} is reached.

Finally the notation $c \prec$ d means that the class c inherits from the class d.

Persistence of an object in an instanciation branch

Let us consider that we want to define a persistent object o_k in the client. According to the chosen persistence paradigm, this may mean either that we want to send a persistent instanciation message to its most specialized class o_{k+1}, in order to create o_k, or that we want to send a replication message to o_k itself. In short, in order to get o_k persistent, we need to send a message $m_{persistence}$, which semantics may be related to the instanciation or to the replication process, to an object o_i, that may be equal to o_{k+1} or o_k according to the semantics of $m_{persistence}$.

In any way, the persistence of o_k implies that, in the server, the class metaobject $(o_{k+1})_{server}$ exists, and is associated to the class metaobject o_{k+1} of the client. So an object $(o_k)_{server} \lhd (o_{k+1})_{server}$ associated to o_k is created in the server.

The object o_{k+1} must have been previously set persistent in order to set o_k persistent. Since o_k is persistent when the message $m_{persistence}$ is sent to the object o_i, the object o_{k+1} is persistent when the message $m_{persistence}$ is sent to the object o_{i+1}.

Management of the instanciation root

Let us assume that o_i belongs to a finite branch of direct instanciation relationships, that is, there exists a class metaobject o_s, meta^{s-i-1}class of o_i, which is instance of itself[1]. By induction on the instanciation relationships, we see that, first, the message $m_{persistence}$ must be sent to o_s, then to o_{s-1}, to o_{s-2}, …, and finally to o_i.

According to this proposal, since o_s is instance of itself, its associated class metaobject in the server $(o_s)_{server}$ should be created by sending to itself an instanciation message. This is obviously a paradox. But, if there is an infinite tower of meta-levels, and if there is neither a cycle, nor a root, on the instanciation branch, the regression in the branch would be infinite and no object would be created on the server.

In order to solve these problems, we assume that the object o_s has not been created by instanciation. The main idea is to consider that, for a given instanciation branch, we have a client and a server that share at least a common class metaobject. This does not mean that we have necessary the same language or model in the client and the server. There may be, in the client, objects and class metaobjects instances of o_s that belong to the language but that are not present in the server. Moreover, it does not mean that o_s must necessary exist in the server. Being able to manage instances of o_s as if $(o_s)_{server}$ exists is sufficient (see, for example [8]). Note that o_s can be any language or user-defined class metaobject instead of an instanciation root.

We sum up the problem we face by the following three points:

- There is, at least a class metaobject o_s common to the client and the server.
- There are transient objects of the client that we cannot change – these objects may be standard metaobjects, or class metaobjects, of the language that are not present in the server, or they may be classes of a user library that cannot be modified.
- There may be objects of the client that can be changed – for example, the classes of an user application being designed.

We show now how persistence libraries can be designed, according to the language properties and the needs of the user.

[1] The class `class` in ObjVLisp corresponds to o_s.

3. DESIGNING THE OBJECT MODEL OF A PERSISTENCE CLASS LIBRARY

3.1. Managing the instances of a natively persistent class

We consider in a first time that we have, in the client, no class metaobject that cannot be changed.

We have stated that a class metaobject o_s is "natively persistent". However this does not imply that all the instanciation sub-branch rooted on o_s can be considered as persistent.

As a matter of fact, let us consider that o_s is the meta$^{s\text{-}i\text{-}1}$class of an object o_i. If we can modify all the class metaobjects o_{i+1}, …, o_{s-1}, we can design on them an instanciation or replication behavior $m_{persistence}$. But we cannot modify o_s itself. So o_{s-1} will not be able to receive the message $m_{persistence}$. If $m_{persistence}$ is a replication message, the object o_{s-1} cannot be replicated. If $m_{persistence}$ is a persistent instanciation message, since it is defined neither on o_s nor on o_{s+1}, the persistent objects o_{s-1} and o_{s-2} cannot be created.

Using the replication paradigm

We propose the following model to manage persistent instances of o_s when using the replication paradigm. We define, in the client, a class Po_s (its name stands for "persistent o_s") with the following properties:

- □ $Po_s \lhd o_s$
- □ $Po_s \prec o_s$
- □ Po_s specializes the behavior $m_{persistence}$ (that is, the replication behavior) with two special semantics: (1) when an instance of Po_s receives the $m_{persistence}$ message, it does not send it to its own class; (2) the $m_{persistence}$ behavior instanciates the $(o_s)_{server}$ class associated to (o_s).

As a consequence of these rules, Po_s has no associated class in the server.

We require that o_{s-1} be an instance of Po_s. We can note that it is still an instance of o_s. When the object o_{s-1} receives the message $m_{persistence}$, the behavior $m_{persistence}$ defined on Po_s is invoked and an instanciation message is sent, in the server, to $(o_s)_{server}$. The object $(o_{s-1})_{server} \lhd (o_s)_{server}$ is created. The only difference between o_{s-1} and $(o_{s-1})_{server}$ is the impossibility to send the $m_{persistence}$ message to $(o_{s-1})_{server}$. Nevertheless, this impossibility is not a problem since the semantics of $m_{persistence}$ is to set persistent a transient object and $(o_{s-1})_{server}$ is already persistent.

Using the persistent instanciation paradigm

We propose a similar approach to manage the persistent instanciation paradigm. We define a similar class Po_s. The only difference is that $m_{persistence}$ is an instanciation message. So there is, of course, no need to send it to the receiver's own class. Moreover we define a class $subPo_s$ which properties are:

- □ $subPo_s \lhd Po_s$;
- □ $subPo_s \prec Po_s$.

When the message $m_{persistence}$ is sent to $subPo_s$ an object $(o_{s-1})_{server} \lhd (o_s)_{server}$ is created.

3.2. Modifying an existing model

When the designer defines a new application from scratch, s/he can create a class p_1 – which notation stands for "persistent object" – and define on it the behavior $m_{persistence}$. Then s/he defines the class Po_s and defines on it the specialization of $m_{persistence}$ that we have proposed.

Finally, s/he can specify her/his own classes $o_1, \ldots, o_{s-1}$ as follows:

- □ $o_{s-1} \lhd Po_s$ or $o_{s-1} \lhd subPo_s$ according to the semantics of $m_{persistence}$;
- □ $\forall i \in [0, s\text{-}2]$, $o_i \lhd p_1$, that is, $\forall i \in [1, s\text{-}1]$, $o_i \prec p_1$.

Since several instanciation levels of a same instanciation branch inherit of a same class p_1, metaclass compatibility problems can arise according to the classes and metaclasses of p_1. We study this problem in the section 5.

3.3. Extending the model using multiple inheritance

The previous model implies that all of the class metaobjects $o_1, \ldots, o_{s-1}$ can be modified. This may be impossible if there are, in this list, some standard class metaobjects belonging to the language, or class metaobjects reused from a class library. In this case, if the language allows multiple inheritance, we propose to define new persistent class metaobjects $\{o'_1, \ldots, o'_{s-1}\}$ inheriting from the transient class metaobjects:

- □ $\forall i \in [1, s\text{-}1]$, $o'_i \prec o_i$ and $o'_i \prec p_1$;
- □ $\forall i \in [0, s\text{-}2]$, $o'_i \lhd o'_{i+1}$;
- □ $o'_{s-1} \lhd subPo_s$ or $o'_{s-1} \lhd Po_s$ according to the persistence paradigm (persistent instanciation or persistent replication).

The user must use the class metaobjects $o'_1, \ldots, o'_{s-1}$ instead of the class metaobjects $o_1, \ldots, o_{s-1}$ and instanciate the class o'_1 instead of o_1. Owing to the object-oriented paradigm, this will cause no type-checking problem. However, the management of the inherited transient slots and methods can be a problem. As a matter of fact, we have to manage a graph of persistent and transient class metaobjects. We discuss this point in the section 4. Moreover, as previously, metaclass compatibilities have to be checked. This common problem is presented in the section 5. An applied example of this approach can be found in the figure 8.

3.4. Copying dynamically a transient object model

The most difficult case to manage is when some class metaobjects cannot be modified and the language does not allow multiple inheritance.

A first proposal would be to subclass each class in $\{o_1, \dots, o_{s-1}\}$ by a class $o'_j \prec o_j$ and to define on each o'_j, $j \in [1, s\text{-}1]$, a behavior $m_{persistence}$ dedicated to the persistence management. However, if there are many classes and many instanciation levels, this is obviously a considerable work. So we propose to rely on the introspective properties of a reflective language.

Let us consider that we have a transient object o_i such that $o_i \lhd o_{i+1}$. We want to create another persistent object o'_i, instance of a class metaobject $o'_{i+1} \prec p_1$. The object o'_i has no link with o_i or o_{i+1}, but it is, semantically, a copy of o_i. The class metaobject o'_{i+1} is itself instance of a copy of o_{i+1}. So any message that can be sent to o_i can be sent to o'_i. It will be executed in the same way on o'_i that it would have been executed on o_i.

The goal is in no way to try to set persistent the class o_{i+1} or the object o_i. We do not pretend that o'_i and o'_{i+1} should be used in the application instead of o_i and o_{i+1}. After all, the type checker of the language would not allow it. But if the designer has a transient application which data and models are important to store and to share, s/he can realize a copy of this model and these data. The identifiers and the actual types of the copied objects and class metaobjects are different than the original ones, but the semantics and the encapsulated data are identical.

This proposal relies on the dynamic and introspective properties of a reflective language: the language may be able to inspect objects, metaobjects and class metaobjects, and to dynamically create classes and metaclasses.

Proposal of an object model

This proposal implies that a copy instanciation behavior can be defined. Let us consider two classes o_i and o'_i, let us assume that the properties of the first one are a subset of the properties of the second one[2] and let us consider an object $o_{i-1} \lhd o_i$. It must be possible to send a copy instanciation message $new(o_{i-1})$ to o'_i in order to get a new object $o'_{i-1} \lhd o'_i$ encapsulating data equal to the data encapsulated by o_{i-1}.

We use two notations:

- ☐ $o'_k \leftarrow new(o_{k-1})$ means that o'_k is instanciated to create a new object, or class metaobject, $o'_{k-1} \lhd o'_k$, semantically equal to o_{k-1}.

- ☐ $o'_k \leftarrow new(o_{k-1}, c)$ means that the metaclass o'_k is instanciated to create a new class metaobject o'_{k-1} inheriting from the class metaobject c, semantically equal to o_{k-1}.

So, if a designer has a persistent instanciation branch $o_0, o_1, \dots, o_{s-1}$ (with $o_{s-1} \lhd o_s$) that s/he wants to duplicate into a persistent instanciation branch $o'_0, o'_1, \dots, o'_{s-1}$, we propose to use the following model.

The classes Po_s and $subPo_s$ are the ones described previously. Moreover, the copy instanciation behavior *new* is defined on Po_s. We assume that the behavior *new* is also defined on a class *CopyClass* inheriting from the persistence class p_1. So, in order to get o'_0, the following messages must be successively sent:

[2] We do not mean that o'_i should inherit from o_i but we require that the selectors of o_i can be found in o'_i with the same types and properties.

□ $subPo_s \leftarrow new(o'_{s-1}, CopyClass)$;

□ for i decreasing from $s-1$ to 3, $o'_i \leftarrow new(o_{i-1}, CopyClass)$;

□ $o'_2 \leftarrow new(o_1, p_1)$;

□ $o'_1 \leftarrow new(o_0)$;

With this model, every object from the set $\{o'_0, ..., o'_{s-1}\}$ can receive the persistence message $m_{persistence}$. We present in the figure 3 a schema summing up this approach. In this figure, the persistence message $m_{persistence}$ is the replication message. Note that this approach has a drawback: each transient object is replicated in a persistent object. This can lead to an important waste of memory. In order to save memory, the primary encapsulated data should be shared between the transient and the persistent objects. This approach should be used only when the others cannot be used.

4. MANAGING TRANSIENT AND PERSISTENT CLASS METAOBJECTS

4.1. Transient and persistent objects

We have presented, in the section 3.3, a way to add persistence properties to an existing object model, relying on multiple inheritance. The main idea was to create a subclass c'_i of a transient class c_i, with an equal semantics, then to use c'_i instead of c_i.

Now we need to ponder on the relationships between persistent class metaobjects and transient ones. In most of the reflective object-oriented languages, the inheritance relationship between two classes finds expression in a reference link between the two class metaobjects associated with these classes. Thus, a persistent class metaobject inheriting from a transient class metaobject is a persistent object referencing a transient object.

A common way to manage persistent objects is to use the paradigm of persistence by reachability (see, for example [4]). When, after a transaction commitment, a potentially persistent object o is stored in the database, every potentially persistent object that can be reached from o through reference relationships, is stored. No transient object can be stored in the database[3].

Since classes are objects, the same management is done for them. If a persistent class c'_i inherits from a transient class c_i, then the class metaobject c'_i will have, in the server, an associated class metaobject $(c'_i)_{server}$ but the referred class metaobject c_i will have no associated object in the server. So, the server class metaobject $(c'_i)_{server}$ has no superclass.

As a consequence, all the slots and the behaviors defined in the class c_i are neither defined in, nor inherited by, $(c'_i)_{server}$. Any object $(c'_{i-1})_{server}$, instance of $(c'_i)_{server}$, supposed to be associated to an object

[3] In this section, we name "transient objects" the objects that are not potentially persistent. They are definitively transient.

c'_{i-1} of the client, does not encapsulate the data encapsulated by c'_{i-1} that have been defined on c_i. In the same way, all the messages corresponding to behaviors defined on c_i, that can be sent to c_{i-1} cannot be sent to $(c_{i-1})_{server}$.

This can be useful in some cases, for example, when a superclass has an obviously transient semantics. However, it cannot be generalized, since the object model proposed in the section 3.3 would be meaningless.

4.2. Inheritance management

Two approaches can solve this problem:

□ redefining in the subclass the slots and behaviors of the superclasses;

□ using introspection to dynamically inspect transient metaobjects.

Specializing transient properties

A first choice can be to let the designer redefine in the persistent subclass, all the slots and behaviors of the transient superclass that s/he wants to be treated as persistent.

Of course, the designer can specialize these slots or behaviors. According to the possible properties of the language meta-level, a transparent rerouting towards the selector of the superclass can also be implemented [7].

Inspecting the transient class metaobjects

A second approach relies on the introspection properties of a reflective language. The main idea is to design the persistence behavior (instanciation or replication) so that, when invoked on a class metaobject c'_i, the inheritance relationship is isolated. Then the inherited transient metaobjects are dynamically inspected. Finally, the direct slots and behaviors of the generated persistent class metaobject $(c'_i)_{server}$ are the set of the direct ones of c'_i and the ones inherited from the transient superclasses of c'_i.

For example, let us suppose that, in the client, the persistent class c'_i defines the direct slots $s_1, \ldots, s_m$ and that the transient superclass c_i of c'_i defines the direct slots $s_{m+1}, \ldots, s_n$. Then, the direct slots of the generated class metaobject $(c'_i)_{server}$ will be $s_1, \ldots, s_m, s_{m+1}, \ldots, s_n$.

The first approach implies that the designer chooses the persistent slots and behaviors of her/his transient classes s/he wants to consider as persistent. S/he has to be aware of persistence contingencies. The second approach implies that all the slots of a transient class becomes persistent.

4.3. Slot types

Problems of type declaration in the server

Some type problems can occur in a client environment where transient and persistent class metaobjects coexist.

Let us assume, for example, that a transient class c_i has a slot s which type is a reference to an instance of another transient class d_i. Let us consider that a persistent class $d'_i \prec d_i$ has been defined. When a persistent class $c'_i \prec c_i$ is defined, the slot s should be specialized in c'_i so that its type be a reference to d'_i instead of a reference to d_i.

As a matter of fact, it is not possible, in the server, to define in the class metaobject $(c'_i)_{server}$ a slot which type is a reference to $(d_i)_{server}$. This last class does not exist in the server since d_i is a transient class. But the class d'_i is persistent, so the associated class $(d'_i)_{server}$ exists in the server. As a consequence, a slot which type is a reference to d'_i can be described in $(c'_i)_{server}$.

However, there may be some cases where the type of the inherited transient slots cannot be specialized. For example, a reflective language is often dynamic: the class d'_i may not have been created yet when the c'_i class is saved.

Another case can appear when the class c'_i is created while many instances of d_i have been created yet. The goal of an object instance of c'_i (that is, semantically, a persistent instance of c_i) may be to refer to direct instances of d_i (transient objects) and to direct instances of d'_i (potentially persistent objects).

Finally, the second model presented in the section 4.2, where the slots and the behaviors of the inherited transient classes are transparently inspected, shows a case where the types of the inherited "transient"

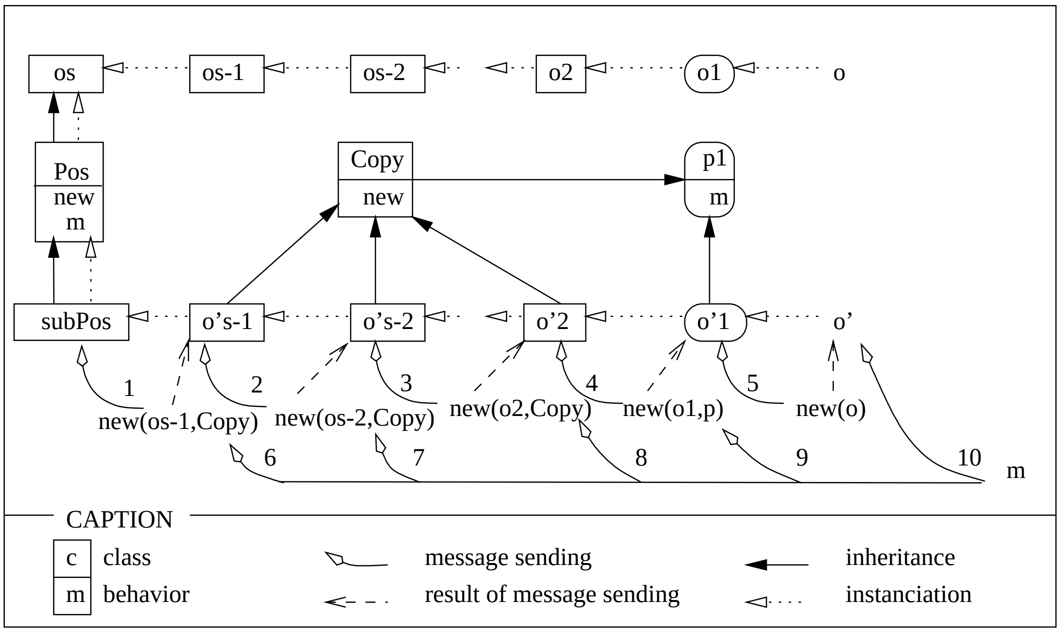

Figure 3. Dynamically copying a transient model.

slots are not specialized.

Setting the slot types in the server

A first answer could be to ignore the slots with a “transient” type. However, this is against the principles of persistence by reachability. A potentially persistent object should be stored when it is refered by a stored persistent object, even if the type of the reference slot is transient. In our example, when an instance of c'_i refers to an instance of d'_i, the latter one should be stored when the former one is stored, whatever the type of the slot s may be.

A second answer could be to let the system automatically generate the persistent subclass of a transient superclass when it cannot find one. However, there is a risk to create incompatibilities with the user object model, since the last one can dynamically evolve too.

So a third possibility can be to loosen, in the server, the types of the transient slots.

If the language has an inheritance root r, like the class `object` in ObjVLisp, and if this inheritance root is common to the client and the server, we can set the type of all the transient slots to r. We symbolize it with the following notation:

☐ $type(c'_i . s) = d_i$ and $type((c'_i)_{server} . s) = r$

Now, let us consider that there is no inheritance root in the language. Our hypothesis throughout this paper is that there is at least a class metaobject c_s common to the client and the server. Moreover, any persistent object that is not a instance of c_s is an instance of the class p_l. So, if the class p_l can be considered as persistent, we can state that[4]:

☐ if $type(c'_i . s) = d_i$, if d_i is transient and if $d'_i \preccurlyeq c_s$, then $type((c'_i)_{server} . s) = (c_s)_{server}$

☐ if $type(c'_i . s) = d_i$, if d_i is transient and $d'_i \not\prec c_s$ then $type((c'_i)_{server} . s) = (p_l)_{server}$

In short, a reference to a subclass of c_s is converted into a reference to $(c_s)_{server}$ and a reference to a subclass of p_l is converted to a reference to $(p_l)_{server}$.

A way to set p_l as persistent is to give persistence properties to the own class of p_l. Provided that the metaclass compatibility problems are solved, it can be done by setting $p_l \lhd Po_s$, or $p_l \lhd subPo_s$ if $subPo_s$ exists.

This looseness of the slot types can obviously generate problems when another designer directly uses the object model stored in the server. If s/he is unaware of the looseness, s/he can link wrong objects together, and s/he will get no type checking warning. However, we present this approach only to solve a problem that can be avoided in most of the cases.

[4] We use the notation $a \preccurlyeq b$ to symbolize that either $a = b$ or $a \prec b$.

5. MANAGING THE METACLASS COMPATIBILITIES

5.1. Importance of the study

In the four generic object models proposed in the section 3, a class is inherited by several classes belonging to different instanciation levels in a same instanciation branch. In this architecture, metaclass compatibility problems can possibly occur.

The metaclass compatibility problem has been presented in [13]. In short, let us suppose that a class c_i is instance of a metaclass c_{i+1} and that a behavior *foo* is defined on c_i while a behavior *metaFoo* is defined on c_{i+1}. Let us assume, finally, that the behavior *foo* is implemented in c_i so that each time an instance of c_i receives a *foo* message, a *metaFoo* message is sent to the own class of the object. For example, when a direct instance c_{i-1} of c_i receives the *foo* message, the *metaFoo* message is sent to c_i. Now, if we subclass c_i by a class d_i, any instance d_{i-1} of d_i should be able to receive the *foo* message. Let us call d_{i+1} the class of d_i. When an object d_{i-1} receives the *foo* message the *metaFoo* message is sent to the own class of d_{i-1}, that is, to d_i. But if d_{i+1} does not define the *metaFoo* behavior, the *metaFoo* message sending fails, and, as a consequence, the *foo* message sending to d_{i-1} fails too.

It may be up to the designer to solve these problems. However, the aim of many systems is to prevent the designer to create metaclass compatibility problems. In the previous example, there cannot be any metaclass compatibility problem if $d_{i+1}=c_{i+1}$ or $d_{i+1} \prec c_{i+1}$. The isomorphic graphs of classes and metaclasses in Smalltalk [12] avoid the compatibility problems. The Ilog Power Classes language [14] enforces that the metaclass of a subclass must be equal to, or be a subclass of, the metaclass of the superclass. The SOM system [6, 11] transparently solves the metaclass compatibility problem by generating adequate metaclasses. However, the metaclasses are connected by an inheritance graph similar to the ones that can be created in Power Classes with multiple inheritance. In [3], the Smalltalk-like inheritance rules between metaclasses are excluded only when a metaclass is designed to manage a unique instance.

So our study would not be complete if we would not take into account the inheritance rules enforced on the meta-levels by several systems with metaclasses: it is not sufficient to state that, for example, the persistent class p_1 should be inherited by k instanciation levels $\{o_1, \ldots, o_k\}$ without considering the inheritance relationships between, on the one hand, each member of the set $\{o_2, \ldots, o_s\}$ and, on the other hand, the own class of p_1, that are induced by this statement in most of the systems. This study is the goal of this section.

5.2. Consequences of the inheritance of a class

We suppose that we have a set of m classes $P = \{p_1, \ldots, p_m\}$ such that $\forall j \in [1, m\text{-}1], p_j \lhd p_{j+1}$, that does not intersect with a second set of n classes $C = \{c_1, \ldots, c_n\}$ such that $\forall j \in [1, n\text{-}1], c_j \lhd c_{j+1}$. If we consider an object $p_0 \lhd p_1$ and another object $c_0 \lhd c_1$, the set P is the maximal set of the meta^{k}classes of p_0 such that no element in P is a meta^{k}class of c_0.

In this paper, p_1 stands for a persistence class and the p_j class metaobjects are its class and metaclasses.

Finally, we assume that the class metaobject c_n is instance of itself, that is, $\forall i > n$, $c_i = c_n$. We enforce that the most specialized class of p_m, that is, p_{m+1}, belongs to the set C. This last requirement means that the set P is a class library of a language. A part of the standard class metaobjects of the language can be found in C.

If the system enforces that the metaclass of a subclass be equal to, or be a subclass of, the metaclass of the superclass, and if a class c_i inherits from a class p_i, then $c_{i+1} \prec p_{i+1}$, and, by induction:

- □ $\forall i \geq 0, c_{i+j} \preccurlyeq p_{i+j}$ (1)

As a consequence of the rule 1, we have:

- □ $\forall i \in [1, m], c_i \prec p_i$ (2)
- □ and, if $m \geq n$, $\forall i \in [n, m], c_n \prec p_i$ (3)

The rule 3, when $m \geq n$, enforces that

- □ either multiple inheritance can be expressed,
- □ or the class c_n inherits from a class $p_q \in \{p_n, \ldots, p_m\}$ that inherits from every class in $\{p_n, \ldots, p_{q-1}, p_{q+1}, \ldots, p_m\}$ by successive single inheritance relationships.

Now, we have to determine the direct class of p_m, that is, p_{m+1}. We have stated that $p_{m+1} \in C$. We have $p_m \lhd c_q$ and $c_q \in C$. Since $c_1 \prec p_1$, we have:

- □ $\forall i \in [1, n\text{-}q], c_{m+i} \preccurlyeq c_{q+i-1}$ (4)

We can note that the rule 4 if true even if $q < m$. We show an example of these relationships in the figure 4 where $m = 4$, $q = 7$ and $n = 10$.

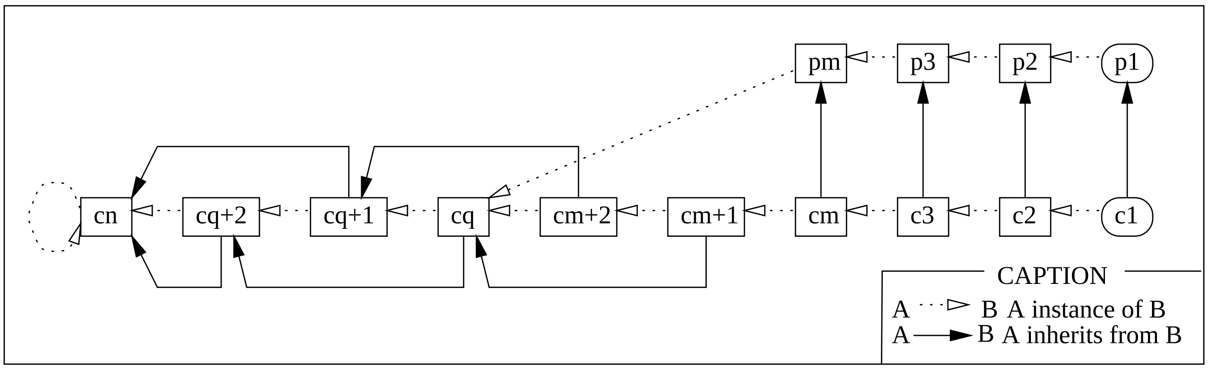

Figure 4. Example of inheritance relationships induced by the inheritance of the class p_1 by the class c_1. Note that p_m and c_m do not share a common metaclass.

Managing the inheritance constraints

We can divide into two sets the constraints presented in the previous discussion.

A first kind of constraint is the connection of elements of the set C to elements of the P set by inheritance relationships. For example, the rules (1), (2) and (3) enforce these connections. These constraints are quite natural: the use of a class library in an application usually implies that the classes of the application inherit from some of the classes of the library.

However, there is another kind of constraints that may raise some problems: the enforcement of inheritance relationships between elements of the set C. The set C contains a subset of class metaobjects belonging to the language itself – at least, the class c_n that is instance of itself. Requiring that these class metaobjects comply with special inheritance relationships means that:

□ either the language itself has to be modified, if these relationships do not exist;

□ or the language will not be able to be changed, if these relationships exist.

So, when designing a class library, it is better not to enforce inheritance relationships inside the set of user-defined class metaobjects or standard class metaobjects. A simple way to do it is enforce that p_m and c_m be instance of a common class, that is,

□ c_m is a user-defined class and $p_m \lhd c_{m+1}$ (5)

5.3. Inheritance of a class by several instanciation levels

The main problem of our study is to allow the inheritance of a class by several instanciation levels. So, let us consider that the k first levels of instanciation in the set C inherit from p_1, that is, $\forall i \in [1, k], c_i \prec p_1$.

The rule 2 implies that

□ $\forall i \in [1, m], \forall j \in [0, k], c_{i+j} \prec p_i$ (6)

and the rule 3 implies that

□ $\forall i \in [n, m+k], c_n \prec p_{i-k}$ (7)

Finally, the rule 4 implies that

□ $\forall i \in [1, n-m-1], \forall q \in [m, m-1+\mathrm{k}], c_{q+i} \prec c_{q+i-1}$ (8)

The important result is that the rule (8) implies no inheritance constraint between members of C if k=1. For example, in the figure 5, the inheritance constraints implied by $k = 2$ are shown.

We have shown in this section that, when using a language that enforces a kind of parallelism between the inheritance tree of a class and the one of its metaclass, whenever more than one instanciation level inherits from a same class, there are inheritance constraints between the own classes and metaclasses of the language and/or the own classes or metaclasses of the application using the library. We have formalized them, so a designer can evaluate if her/his language can express a library usable by several instanciation levels.

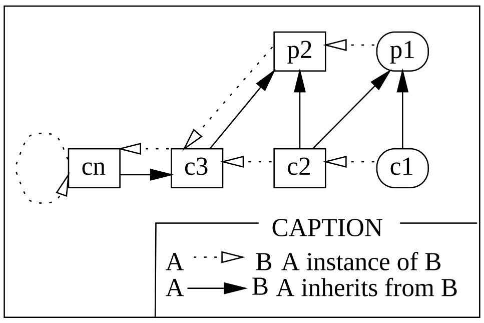

Figure 5. Example of inheritance constraints implied when two instanciation levels inherit from a same class.

5.4. Isolating the instanciation levels

However, there is an expensive way to allow *k* instanciation levels to share some common behaviors, with no inheritance constraints in the set *C* of standard classes and user-defined classes.

We can define *k* classes $p_{1,1}$, …, $p_{1,k}$, with no link among themselves and no link with p_1, implementing all of the behaviors of p_1. Of course, it may be expensive since the code related to persistence management must be duplicated *k* times on $p_{1,1}$, $p_{1,2}$, …, and $p_{1,k}$.

Then we state that:

- □ $\forall i \in [1, k], p_{1,i} \lhd p_{i+1}$

- □ $\forall i \in [1, k], p_i \prec p_{1,i}$

The inheritance constraints induced by $c_1 \prec p_1$ are:

- □ $\forall i \in [1, k], c_i \prec p_i \prec p_{1,i}$

Thus, *k* instanciation levels will have the same persistence behaviors and no inheritance constraints are set in *C*.

The figure 6 shows an example where $k = 2$, $m = 4$, and $n = 6$.

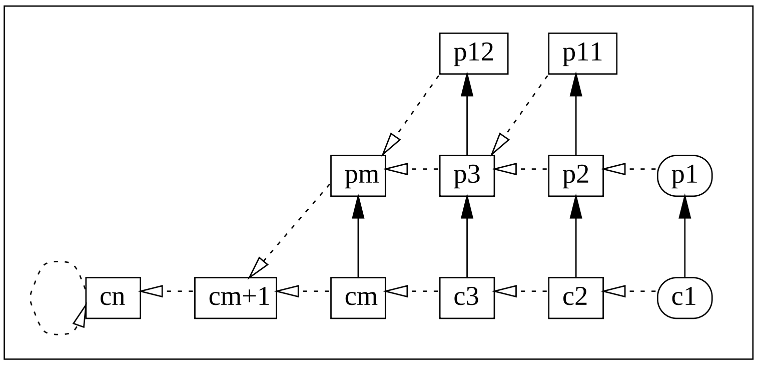

Figure 6. Duplicating the persistence management class.

6. A CASE STUDY: ADDING PERSISTENCE TO THE *POWER CLASSES* LANGUAGE

6.1. Overview of Power Classes

We have applied our proposal to a real case. We provide a persistence library to the Lisp-based reflective and object-oriented language Ilog Power Classes [14]. It is a commercial and enhanced version of EuLisp [21].. The actual persistence tool used is a relational database management system accessed through ObjectDRIVER, our object wrapper [18].

Power Classes has some interesting properties that justify the use of several of our proposals.

- The language deals with possible metaclass compatibility problems by enforcing the rules described in the section 5.1, that is, the metaclass of a subclass must be equal to, or must inherit from, the metaclass of the superclasses.

- The language offers two object model paradigms. In the first one multiple inheritance can be used, in the second one, only simple inheritance can be used.

As we can see in the figure 7, Power Classes offers two standard metaclasses: `standard-class` and `structure-class`. The classes that are instances of `standard-class` can be related using multiple inheritance, while the classes that are instances of `structure-class` can only be related using simple inheritance.

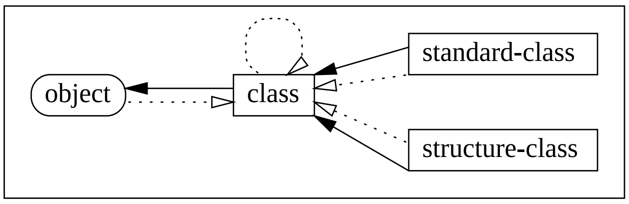

Figure 7. Part of the standard model of Power Classes

The rules enforced to avoid any metaclass compatibility problem forbid the designer to link instances of `structure-class` and `standard-class` by inheritance relationships.

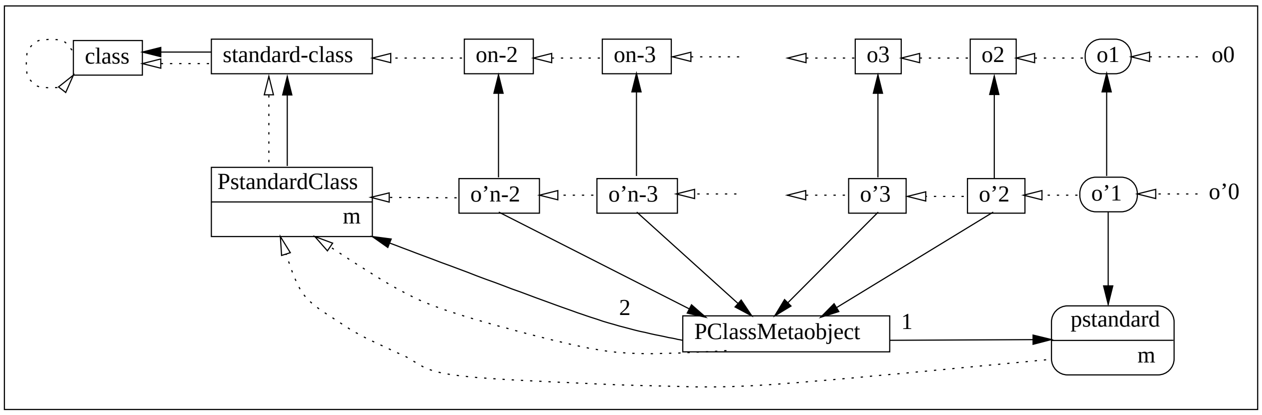

Figure 8. Persistence management in Power Classes using the generic object model of the section 3.3

The possibility to define multiple inheritance for instances of `standard-class` justifies the use of the model based on the multiple inheritance paradigm that has been described in the section 3.3. The impossibility to use multiple inheritance for instances of `structure-class` justifies the dynamic copy of a transient model into a persistent one, as described in the section 3.4. Finally, the rules enforced to ensure metaclass compatibility are compatible with the discussion of the section 5. So, it occurred to us that Power Classes is a good test case for our proposals.

We arbitrarily choose the persistent replication paradigm. An object is at first transiently created in the client, then it is duplicated in the server when it receives a replication message $m_{persistence}$.

In order to bind a reflective object-oriented language with a relational formalism, we used the patterns proposed in this research report. The implementation keeps the resulting language open. This implementation is beyond the scope of this research report and can be found in [7, 8].

6.2. Extending a transient model

In this section, we apply the model proposed in the section 3.3. We study an instanciation branch $o_1, \ldots, o_n$ where o_n is an instance of itself. In Power Classes,

□ the class metaobject o_n is the class `class`;

□ the class metaobject o_{n-1} is the class `standard-class`;

□ moreover, we have $o_{n-1} \prec o_n$, that is, `standard-class` $\prec$ `class`.

As a standard metaobject of the language, we assume that `standard-class` is natively present in the server. It corresponds to the class metaobject o_s of the section 3.1.

We define the metaclass *PstandardClass*, corresponding to the class Po_s. We have

□ *PstandardClass* $\prec$ `standard-class`, and

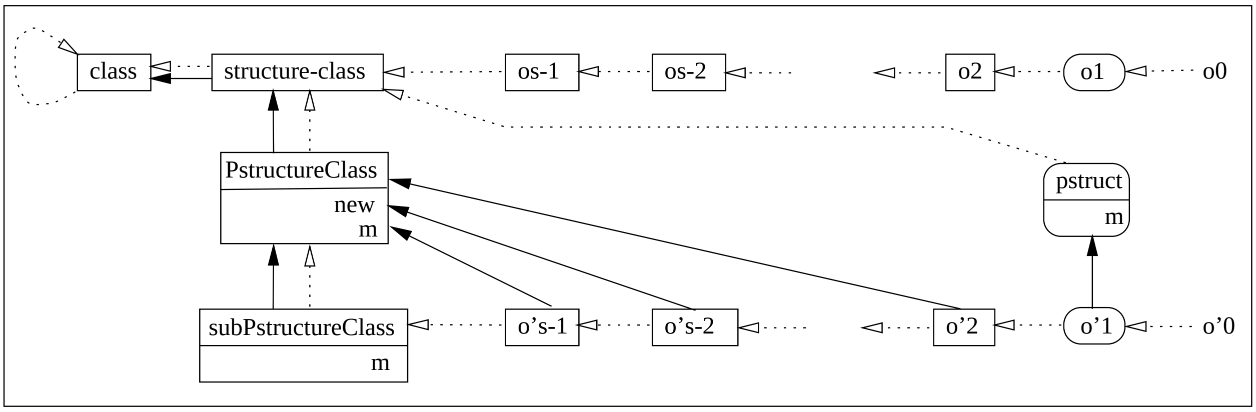

Figure 9. Persistence management in Power Classes using the generic object model of the section 3.4

- $\square$ *PstandardClass* $\triangleleft$ **`standard-class`**.

We define on *PstandardClass* a non-regressive replication message $m_{persistence}$.

Now, we define a class $p_{standard}$, equivalent to the class p_1 of the section 3. We define on $p_{standard}$ the behavior $m_{persistence}$.

We want to define the persistent class metaobjects $o'_1, \ldots, o'_{n-2}$ such that $\forall i \in [1, n\text{-}2]$, $o'_i \prec o_i$ and $o'_i \prec p_{standard}$.

So, n-2 instanciation levels can inherit from $p_{standard}$ but we are only sure that $n \geq 3$.

With regard to the metaclass compatibility problem, since we do not want to change neither **`standard-class`** nor **`class`** (that is, neither o_{n-1} nor o_n), the rule (7) presented in the section 5.3 implies that $p_{standard}$ be an instance of a standard metaclass of the language.

Many applications defined in Power Classes do not define new metaclasses. In these applications, every class is instance of **`standard-class`**. To respect the rule (5), we set $p_{standard} \triangleleft$ **`standard-class`**.

The rule (8) enforces that:

- $\square$ $\forall i \in [2, n\text{-}2]$, $o_i \prec o_{n-1}$

- $\square$ $\forall i \in [2, \text{n-1}]$, $\text{o}_\text{i} \prec o_n$

Now, in Power Classes, $o_{n-1} \prec o_n$ is true. So, the only constraint enforced is that any persistent metaclass defined by the user should inherit from **`standard-class`**. We have noticed that, in most of the cases, this is a weak constraint, since **`standard-class`** provides very useful behaviors and tools.

Since we need the user-defined persistent metaclasses to inherit from **`standard-class`** and from $p_{standard}$, we define the class *PClassMetaobject* such that:

□ *PClassMetaobject* is instance of `standard-class`,

□ *PClassMetaobject* $\prec$ `standard-class`

□ *PClassMetaobject* $\prec p_{standard}$.

The last statement allows to send persistence messages to the objects o'_0, …, o'_{n-3}. The persistence behaviors that can be sent to o'_{n-2} are defined on *PstandardClass*.

We ask the user to define her/his persistent metaclasses as subclasses of *PclassMetaobject* and her/his persistent classes that are not metaclasses, as subclasses of $p_{standard}$[5].

We have seen in the section 4.3 that, in order to manage the types of slots, it can be useful to set persistent the class $p_{standard}$. So we set:

□ $p_{standard} \lhd$ *PstandardClass*

The rules enforced by Power Classes to avoid any metaclass compatibility problem imply that

□ *PClassMetaobject* $\lhd$ *PstandardClass*

□ *PClassMetaobject* $\prec$ *PstandardClass*

□ The persistence behaviors defined on $p_{standard}$ override on *PClassMetaobject* those defined on *PstandardClass*. This can be done by specifying an order in the inheritance tree.

The figure 8 sums up the object model.

6.3. Persistent copies of transient models

In order to manage the classes instances of `structure-class`, that cannot be combined with multiple inheritance, we use the model defined in the section 3.4.

□ We define a class *PstructureClass*, subclass and instance of `structure-class`. We specialize on it the instanciation message as described in the section 3.4.

□ We define a class *subPstructureClass*, subclass and instance of *PstructureClass*. We define on it the non-regressive replication message $m_{persistence}$.

Since we did not want to rewrite the instanciation message we merged the class *CopyClass* and the class *PstructureClass* and we defined the regressive persistence behavior $m_{persistence}$ on *PstructureClass*.

[5] As a side effect, this discrimination between persistent class metaobjects and persistent final instances is useful to manage a non object-oriented server DBMS [8].

However, the classes that are not metaclasses cannot inherit from *PstructureClass* which is a metaclass. So we define another class $p_{structure}$ and we define on it the same persistence behaviors that have been defined on *PstructureClass*. Note that the class *PstructureClass* cannot inherit from $p_{structure}$ since it inherits from `structure-class`, and, as an instance of `structure-class`, it cannot use multiple inheritance.

The figure 9 sums up this model.

7. RELATED WORKS

Few works have explicitly considered the problem of designing persistence libraries for a transient, object-oriented and reflective language, as we did.

Due to the nature of the involved programming languages, most of the bindings between an object-oriented language and a DBMS do not consider the problems induced by a user-defined meta-level. For example, the Java language has metaobjects but, since the user cannot modify them, and cannot add new metaclasses, the management of several levels of instanciation is never needed.

In the reflective OODBMSs Vodak [17] and Tigukat [24], the problem of object persistence has not been considered as we did. The main idea was to offer an object-oriented reflective model useful in the domain of OODBMSs. But, as far as the authors know, there is no study on the coexistence of transient and persistent objects, the reengineering of existent transient models, or the way to provide persistence to an existing reflective model by adding a class library. The whole language is supposed to be natively persistent, possibly by system programming.

The studies the closest to our interests have been done with PCLOS [22, 23]. PCLOS binds the CLOS programming language and a virtual relational DBMS. However, if the final instances and the user-defined classes are managed, the way an arbitrary number of instanciation levels could be managed is not shown. Moreover this study is very bound to the CLOS programming language. We have taken a much more general approach: the Power Classes language is for us only a test case.

8. CONCLUSION

In this research report, we have focused on the design of a class library for persistence management in an object-oriented reflective language. We have shown that persistence management, in such a language, can be, and should be, modeled by an object-oriented model.

So, our goal was to provide to the user design patterns to build persistence libraries in reflective languages. The class-based object-oriented reflective systems have an interesting property: objects can belong to different instanciation levels and be related by instanciation links. This property cannot be found in classical object-oriented programming languages. So, we took this problem into account. We took a general approach, and independently of any programming language, we have proposed different object models, or design patterns, to design such a library, according to:

- □ the persistence paradigm chosen – use of proxies, or replication mechanism;
- □ the needs of the designer – for example, the freedom to create from scratch a new application or the need to add persistence abilities to an existing application;

□ the properties of the language – single or multiple inheritance, introspection properties, rules to avoid metaclass compatibility problems.

According to the set of her/his needs and her/his constraints, the designer who wants to design persistent application exploiting different instanciation levels, can choose one of our object models: s/he can see the reasons of its use, its interests or its drawbacks. One of the interest of our work is to specifically deal with multiple instanciation levels: the designer does not have to extend a design pattern managing only the instanciation level of the user-defined classes.

When an adequate design pattern has been chosen, the designer can then choose the storage tool s/he prefers without any change in the object model. For example, s/he can use a direct storage in an OODBMS, or an heterogeneous storage as we did with the language Power Classes.

REFERENCES

1. D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales and D. Moon, Common Lisp Object System Specification, in *SIGPLAN notices 23 (special issue)*, 1988.

2. D. Bobrow, R. Gabriel and J. White, CLOS in Context, in *Object-Oriented Programming: the CLOS Perspective*, the MIT Press, chap. 2., pp. 29-61, 1995.

3. N. Bouraqadi-Saâdani, T. Ledoux and F. Rivard, Safe Metaclass Programming, in *Proceeding of the Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA'98*, 1998.

4. R. Catell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Pringer, H. Strickland and D. Wade, *The Object Database Standard: ODMG 2.0*, Morgan-Kauffman, 1997.

5. P. Cointe, Metaclasses are First Class: the ObjVLisp Model, in *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA'87*, pp. 156-167, 1987.

6. S. Danforth and I. Forman, Reflections on Metaclass Programming in SOM, *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'94*, pp. 440-452, 1994

7. S. Demphlous, *Gestion de la persistance au sein de systèmes réflexifs à objets*, Ph.D. Thesis, University of Nice-Sophia Antipolis, France, 1998.

8. S. Demphlous and F. Lebastard, Designing Persistence Libraries in Reflective Models with Intercession Property for a Client-Server Environment: RDBMS management, to appear in *2^d Intl Conf. on Meta-level Architecture and Reflection (Reflection'99)*, 1999.

9. O. Diaz and N. Paton, Extending OODBMSs Using Metaclasses, in *IEEE Software*, vol. 11(3), pp. 28-39, 1994.

10. J. Ferber, Computational Reflection in Class based Object Oriented Languages, *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'89*, pp. 317-326, 1989.

11. I. Forman and S. Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1998.

12. A. Goldberg and D. Robson, *Smalltalk-80: the Language and its Implementation*, Addison Wesley, 1983.

13. N. Graube, Metaclass Compatibility, *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'89*, pp. 305-315, 1989.

14. Ilog, *Ilog Power Classes Reference Manual, version 1.3*, Gentilly, France, 1994.

15. G. Kiczales, J. des Rivières and D. Bobrow, *The Art of the Metaobject Protocol*, the MIT Press, 1991.

16. W. Klas, G. Fischer and K. Aberer, Integrating Relational and Object-Oriented Database Systems using a Metaclass Concept, in *Journal of Systems Integration*, vol. 4, pp. 341-372, 1994.

17. W. Klas and M. Schref, Metaclasses and their application: Data Model Tailoring and Database Integration, *Lecture Notes in Computer Science n. 943*, Springer Verlag, 1995.

18. F. Lebastard, S. Demphlous, V. Aguiléra and O. Jautzy, *ObjectDRIVER Reference Manual*, `http://www.inria.fr/cermics/dbteam/ ObjectDriver`, 1999.

19. P. Maes, Concepts and Experiments in Computational Reflection, *Proceedings of the Conference on Object-Oriented Programming, Languages, Systems and Applications, OOPSLA'87*, pp. 147-155, 1987

20. Poet Software, *Poet Technical Overview*, technical report, `http://www.poet.com/techover`, 1998

21. J. Padget, G. Nuyens and H. Bretthauer, An Overview of EuLisp, *Lisp and Symbolic Computation*, vol. 6, N. 1/2, pp. 9-99, 1993.

22. A. Paepcke, PCLOS: A flexible Implementation of Clos Persistence, *Proc. of ECOOP, Lecture Note in Computer Science n. 322*, pp. 374-389, Springer-Verlag, 1988.

23. A. Paepcke, User-level Language Crafting, in *Object-Oriented Programming – the CLOS Perspective*, the MIT Press, chap. 3, pp. 65-101, 1993

24. R. Peters, *Tigukat: a Uniform Behavioral Objectbase Management System*, Ph.D. thesis, University of Alberta, Canada, 1994.