LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

REVERSE INHERITANCE: AN APPROACH FOR MODELING ADAPTATION AND EVOLUTION OF APPLICATIONS

Ciprian-Bogdan Chirila, Pierre Crescenzo, Philippe Lahire

Projet OCL

Rapport de recherche
ISRN I3S/RR–2005-05–FR

Février 2005

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

Dans le but de faciliter l'adaptation, l'évolution voire la restructuration des applications écrites dans un langage à objets, nous proposons une approche basée sur l'utilisation d'une relation d'héritage inverse.

Nous étudions la modélisation de cette relation, qui est particulièrement adaptée à la restructuration des hiérarchies de classes, les apports et limites de cette technique et les mécanismes à mettre en oeuvre.

Nous proposons un exemple d'extension syntaxique de Java pour illustrer notre proposition.

MOTS CLÉS :

héritage inverse, langages à objets, adaptation, évolution, restructuration

---

ABSTRACT:

In order to facilitate adaptation, evolution, and software re-engineering of application which are written with an object-oriented language, we propose an approach bases on a reverse inheritance relationship.

We study the modeling of this relationship, the benefits and limits of this approach, and the ways and means to apply it.

We propose an example of a Java syntax extension to illustrate our proposition.

KEY WORDS :

adaptation, evolution, re-engineering, reverse inheritance, object-oriented programming languages

Reverse Inheritance : An Approach for Modeling Adaptation and Evolution of Applications

Ciprian-Bogdan Chirilă[*], Pierre Crescenzo[**], and Philippe Lahire[**]

[*] *Faculty of Automatics and Computer Science*
"Politehnica" University of Timişoara
Bd. V. Parvan no 2, 1900 Timişoara
Romania
Chirila@cs.utt.ro

[**] *Université de Nice-Sophia Antipolis*
Laboratoire I3S (UNSA/CNRS)
Projet OCL
2000, route des Lucioles
Les Algorithmes, Bâtiment Euclide
BP 121
F-06903 Sophia Antipolis cedex France
{Pierre.Crescenzo,Philippe.Lahire}@unice.fr

RÉSUMÉ. *Dans le but de faciliter l'adaptation, l'évolution voire la restructuration des applications écrites dans un langage à objets, nous proposons une approche basée sur l'utilisation d'une relation d'héritage inverse. Nous étudions la modélisation de cette relation, qui est particulièrement adaptée à la restructuration des hiérarchies de classes, les apports et limites de cette technique et les mécanismes à mettre en œuvre. Nous proposons un exemple d'extension syntaxique de Java pour illustrer notre proposition.*

ABSTRACT. *In order to facilitate adaptation, evolution, and software re-engineering of application which are written with an object-oriented language, we propose an approach bases on a reverse inheritance relationship. We study the modeling of this relationship, the benefits and limits of this approach, and the ways and means to apply it. We propose an example of a Java syntax extension to illustrate our proposition.*

MOTS-CLÉS : *héritage inverse, langages à objets, adaptation, évolution, restructuration*

KEYWORDS: *adaptation, evolution, re-engineering, reverse inheritance, object-oriented programming languages*

1re soumission à *rapport de recherche I3S*, le 4 février 2005.

1. Introduction

The aim of this paper is to propose an approach which favours both the handling of application evolution and the adaptation of library of classes to the need of a given application. This approach is based on the definition of a new relationship which is called **reverse inheritance**. Main aspect of this approach is to use reverse inheritance in order to specify most of the changes of applications. It is important to understand that if inheritance relationship is not dealing only with specialisation [CRE 03], reverse inheritance is also not only dealing with generalisation.

To adapt a library of classes (whose source-code may not be accessible) to the specific need of an application is not the same process as to have the source code of one application and to make it evolve in order to issue new requirements. The use of both reverse-inheritance and inheritance should allow the programmer to address both objectives, so that it may maximise the reuse of classes.

A study of the state of the art related to re-engineering convinces us that effectively, most of the capabilities needed to handle the evolution and adaptation of hierarchies of classes may be implemented through reverse inheritance relationships. Some of these capabilities among others are the adaptation of the behaviour of components based on classes [HEI 98a], the replacement of unwanted characteristics with suitable properties [CAS 95], or the modification of the class interface [HEI 98b]. In this paper we do not provide an approach for the propagation of changes in the library of classes, neither we provide any versioning mechanism but we propose a support for the description of those changes with the idea that it will make easier the automatizing of their propagation when a library is reorganised.

It is important to note that we are mainly interesting by the expressiveness of reverse inheritance according to the description of the re-engineering capabilities ; other approaches are presented in section 4. It is not our objective to oblige the programmer (which is in charge of evolution or adaptation), to write source code including reverse inheritance relationships even if it is one possibility among others. In particular those changes could be specified thanks to a dedicated wizard and possible views of the library (generated by such wizard) may include source code using both inheritance and reverse inheritance.

Moreover, inheritance relationships must remain the main relationship whereas reverse-inheritance should only be transient (between two reorganisations of the library), or used for implementing the specific needs of a given application. We intend to propose an implementation built on top of a generator which thanks to the information given through the use of reverse-inheritance relationship may produce an equivalent code with only inheritance relationship. In the following we propose an intuitive description of the basics mechanisms of reverse-inheritance relationship which is as much as possible, independent from any language. But we chose also to provide the reader what an extension of Java which support reverse-inheritance could be. The use of annotation as it is proposed for inheritance in [CRE 03] should help to control the use of reverse inheritance in order to ensure that the generator produces automati-

cally a readable re-factored application which addresses only the native inheritance relationship(s).

Second section proposes case studies about some of the most common situations that a programmer may face during both adaptation and evolution processes. It will give the reader a flavour of possible changes that may be handled by reverse inheritance. In the third section we propose some of the basic mechanisms that should equip reverse inheritance in order to handle the changes proposed in section 2. The fourth section gives some related works dealing with the reuse and the maintenance of libraries of classes. Finally we conclude and mention future work.

2. Overview of the Approach through Case Studies

In the following we propose some case studies that point out the re-engineering capabilities of reverse-inheritance. In particular, our examples insist on how a reverse inheritance relationship can bring solutions for the description of changes within the source-code. Each of the next subsections describes one case study which stands for a specific functionality. each of them is described by an UML class diagram in order to be independent from any object-oriented programming language. It will be interesting also, to point out possible problems underlined by the use of reverse inheritance.

Case-studies are presented in order to show the continuous evolution (step by step and time after time) of a library of graphical objects. In following examples we will sometimes make the assumption that part of this library may not be changed by the people in charge of the maintenance. This library starts with very specific items and is is refined case-study after case-study in order to reach a more reusable modeling.

2.1. *To share common functionalities*

To extend a hierarchy through the use of inheritance implies the creation of specialised classes whereas extending the library with reverse inheritance means the creation of more abstract classes. In programming languages such as Java or Eiffel, the inheritance relationship may be used for the specification of both specialisation and generalisation relationships [ARN 00]. In UML specialisation is implemented using generalisation relationship [GRO 03] ; although the two relationships are redundant they may be used together [CRE 02]. The programmer may achieve both adaptation and evolution of a given library through reverse inheritance relationship. In this case such use of this relationship will have the behaviour of one generalisation. Let us suppose that a CAD/CAM software designer uses rectangles, ellipses and triangles within his application (see figure 1). In order to represent them the software designer describes one class for each concrete shape type ; they are called : *Rectangle*, *Ellipse* and *Triangle*. These classes belong to a library of classes proposed by third-party vendors or, are part of a legacy system that must not be touched. The handling of those shapes at a more abstract level is necessary as soon as common functionalities are

found in several classes and have to be handled globally. A well known technique is to create a class which abstract these common functionalities. It becomes an ancestor of corresponding classes.

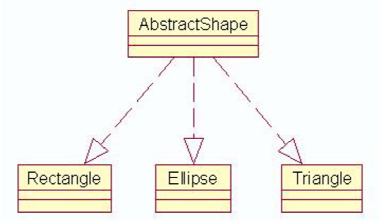

Figure 1. *To capture common functionalities*

The code which is common to all shapes must be moved to the class *AbstractShape* in order to avoid code duplication. The use of reverse inheritance relationship allows to deal with those constraints (in particular the fact that existing class source-code may not be updated) and preserves code readability.

2.2. *Inserting a Class into an Existing Hierarchy*

This case study (see figure 2), deals with the modification of the hierarchy of classes mentioned above, through the addition of a class which handles an intermediate level of abstraction not considered at the moment. This addresses typically the adaptability [LIE 95] of the library which must be easily updated in order to take into account some leaks in the design. We intend to show through following example that hierarchy re-engineering may be solved by reverse inheritance. For some reasons, in the library of graphical objects which is proposed, the classes named Parallelogram and Square are read-only, but we need to add a new class to the hierarchy which preserves the architecture consistency [CRE 02]. In Euclidean geometry any rectangle is a parallelogram, any square is a rectangle ; this forces type Rectangle to be a subtype of Parallelogram as well as a super-type of Square. First constraint may be implemented very easily with classic inheritance relationship. The second constraint could be solved using also a classic inheritance relationship between Rectangle and Square. But this is possible only if Square is not read-only. Otherwise we may use the fact that type Rectangle is a generalisation of type Square ; so that we can use reverse inheritance relationship in order to insert Parallelogram into the hierarchy.

2.3. *Extending an Existing Hierarchy*

This case-study deals with the extension of one hierarchy with third party software. Right now, the graphic library contains only types Parallelogram and Square but some new needs make necessary the inclusion of new types of graphical object : Ellipse

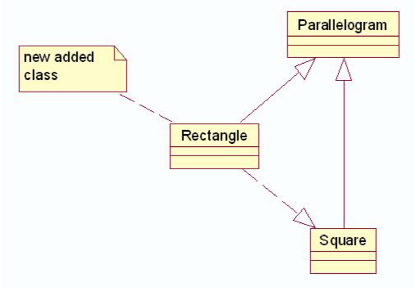

Figure 2. *To insert a Class into a Hierarchy.*

and its specialisations. This may be implemented by the creation of a new abstract type called Shape which becomes the super-type of both Parallelogram and Ellipse. Class *Ellipse* is derived from class *Shape* whereas class *Circle* is derived from class *Ellipse*. Let us consider how to address this hierarchy extension if the constraint is to not change class Parallelogram. Again type Shape should indicate that it generalises type Parallelogram ; so that reverse inheritance can be used to link those two classes (see figure 3).

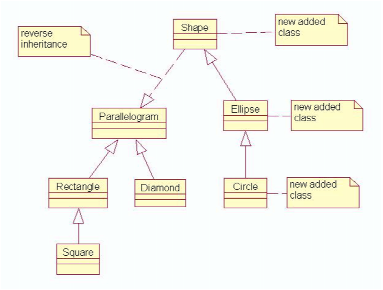

Figure 3. *To extend an existing hierarchy.*

2.4. *Adding Features to Classes*

This case-study deals with the need to add new functionalities, new pieces of implementation or new information to a class or a library of classes. An interesting issue is to ensure that the source-code of existing classes is not changed. Let us suppose that it misses one attribute and its accessors which deal with the description of the background colour of the graphical objects. We propose to refine our graphical library and to use again reverse-inheritance in order to add the class *AbstractShape* which contains the new features. It is important to note at this stage that name conflicts may arise because of the addition of functionalities. We come back on this aspect in next section and we propose some answers in section 3.

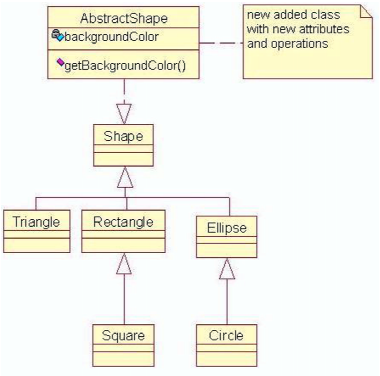

Figure 4. *To add Features to Classes.*

The interest to add the functionalities using reverse inheritance is that it allows (in the same way as classic inheritance) the incremental addition of new services but it does not oblige to create new sub-hierarchies that would lead to a very complex library if the class to update has already a deep sub-hierarchy [LAH 03].

2.5. *Factoring Features from Classes*

Another possible issue is to factorise as much as possible common functionalities from hierarchies of classes. This is very important in order to avoid data and code duplication and to facilitate software maintenance [MAR 00]. In particular [GOD 93] proposes to rely on a Galois Lattice structure in order to perform factorisations as well as other changes ; it will be interesting to look how we may take advantage of it in the implementation of our approach. Processes of factorisation as well as of adding new classes may generate name conflicts. Some answers according to this problem are proposed in [LAW 94] and [SAK 02] in the context of reverse inheritance relationship. We will take them into account when designing both factorisation and renaming capabilities.

In this case-study (see figure 5), we deal with a couple of classes that have some functionalities which are the same ; this is because they have been designed by different people, independently. They must be reorganised in order to get a library with better skills. The functionalities may correspond to methods with different names or signatures ; but in our example we consider a simpler situation where names and signatures are the same in all classes. All the other situations can be reduced to it through the use of the renaming capability (see section 3). Concretely in our example, each graphical object defines a method *redraw()* ; it has to be factorised. Some classes in a hierarchy may have a concrete implementation of a feature and some may not. It is necessary to think about the possible situations that may appear : what happens when in some classes the method has a different body or when it is abstract ? Let us consi-

der that class Triangle has no implementation for redraw() : to factorise this method allows for example, to fill it and to propagate its behaviour to descendants.

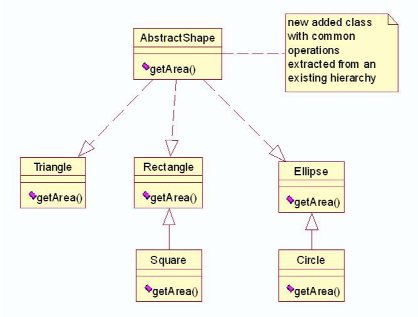

Figure 5. *Factorisation of features*

The use of renaming capabilities can favour also the process of factorisation ; for example, it may be used to rename a feature with the same functionality and different name. Let us consider that class *Square* has another name for the method which redraws the object : *repeat_display ()* ; it must be renamed as *redraw()*.

3. Basic Capabilities of Reverse Inheritance

Like classical inheritance, the reverse inheritance relationship may be used to handle several situations [MEY 02]. In this section we present an overview of the basic mechanisms. They give a flavour of the expressiveness of reverse inheritance for the specification of changes in a hierarchy of classes. They do not intend to address all problems dealing with evolution but they aim to contribute to a more easier reorganisation and extension of a library of classes. For each mechanism we give an intuitive description of its behaviour and we outline some aspects of its implementation.

3.1. *What Kind of Reverse Inheritance*

To introduce reverse inheritance relationship arises a certain number of question that must be fixed :

– Should a class which uses a reverse inheritance relationship may use also classical inheritance relationship(s) ?

– May a class be linked to several classes through reverse inheritance relationship ? In other word, should reverse inheritance be single or multiple ?

– . . .

To answer “yes” to the first question means to increase the expressiveness of the approach and the reusability of the targeted library. If we look to the second case-study

(see figure 2), it is clear that to be able to use also classical inheritance is crucial. If not it implies that we loose an information (in this example : the fact that a *rectangle* is also a *parallelogram*). On the other hand, as it is pointed out in **??** and in [SAK 02], to allow both types of inheritance within the same class introduces many problems to solve according to the targeted language (name conflicts, consistency of assertions, etc). To answer "no" to the second question makes meaningless the use of reverse inheritance as it will be outlined in section 3.3. On a more general point of view the answers to these questions are very dependent from the targeted language. This is particularly true if we deal with code transformation in order to keep only classical relationship after re-engineering : what will be the readability of the code if we are in the context of a language which implements single inheritance such as Java ?

3.2. *Feature Adding Mechanism*

This mechanism intends to provide the same facilities as those provided by classical inheritance for adding features within a class ; we propose here the key-aspects. The intuitive semantics of feature addition through the use of reverse inheritance is "When one class which is inserted in the hierarchy through reverse inheritance declares a new feature, it (attribute or method) belongs to the class as any other". This very naive definition hides a certain number of problem that must be carried out. One of the most important is related to the fact that it may introduce name conflicts in the descendants [SAK 02]. It should be handled in a different way whether features are used for the internal implementation of the class or for inserting a new functionality exported to other classes (descendants or clients). For example, a method which matches the first situation should not arise any conflict because it is not seen by other classes. This aspect related to possible conflicts and to the visibility of a feature will be handled differently depending on the targeted language (Eiffel, Java, etc.). Feature-adding mechanism has to take into account also various other contexts. For example, when the feature is an attribute, is it a class variable or an instance variable ? When a method exists in one descendant of the class with the same signature, should it be considered after the addition of the feature as a redefinition or as a completely different feature ? All these questions must be covered by the chosen solution.

3.3. *Factorisation Mechanism*

To be able to factorise features, whatever they are attributes or methods, is an important aspect of the maintenance of an application [DIC 96, GOD 02]. The interest of this mechanism is shown especially in the fifth case-study (see figure 5) and its main objective is to avoid code duplication. We can summarise the effect of factorisation by following sentence : "When one class which is inserted in the hierarchy through reverse inheritance factories the definition of one feature, this leads to put its definition in the class ; all other definitions may be deleted or not, depending on the context". To factorise a method described in several classes does not lead to any name conflict

but it may lead to behaviour conflicts. For example, all the implementation of the feature to be factorised may not be the same, or it may not be implemented in some descendants (it is typically the case for deferred features in Eiffel or abstract methods in Java), or else a descendant may not yet include this feature. The expressiveness of the factorisation process must take this into account and must allow to consider whether one version of a factorised feature in a descendant class is a redefinition or if it must be forgotten. This is not a strait forward issue because the descendant class should not be modified. Another important aspect about factorisation is that people should be able to distinguish the definition of a factorised feature from the addition of a new one.

3.4. *Descendant Access Mechanism*

If we look to the classical inheritance relationship, we notice that most programming languages provide - to a redefined feature - some way to refer to its original code in an ancestor class. It is present in particular in Eiffel, C++ or Java through keywords such as *: :* in C++, *super* in Java or *precursor* in Eiffel. It avoids code duplication and it is particularly useful when in the subclass some code from the superclass has to be reused. Following the same idea we think that it is also helpful when classes are linked through reverse inheritance relationship : some mechanism must be supplied in order to refer to the redefinition of a feature in descendants. Because reverse inheritance accept multiple targets, it is also mandatory to be able to address one particular descendant.

3.5. *Renaming Mechanism*

The capability to rename a feature is found in programming languages such as Eiffel [MEY 02]. We think that it is very needed in order *i)* to make available to possible client-classes, a name which is more suitable to the local context ; *ii)* to handle possible name conflicts : two features having the same name may not address always the same functionality. In [SAK 02] the author distinguishes two kinds of name conflicts : one happens when the names of two features are equal but they are different methods whereas the second one is when their names are different, but it is the same method. Typically feature renaming will be used in the context of any of the mechanisms mentioned above.

3.6. *Possible Syntax for Java*

We propose below some Java code which rely on a possible Java extension integrating reverse inheritance. It uses classes from the example described in section 2. We propose to extend the class hierarchy using reverse inheritance.

```
class Parallelogram {
 private int perimeter;
 private int area;
}

class Ellipse {
 private int perimeter;
 private int surface;
 public void getSemiPerimeter()
 {
    return perimeter/2;
 }
}

class Shape infers Parallelogram, Ellipse //Reverse inheritances
{
 factored protected int perimeter;        //Feature marked as factored
 protected int color;                     //New added feature
 factored protected int area=
    {Parallelogram.area, Ellipse.surface}; //Renamed feature
 public void getSemiPerimeter(){
    inferior(Ellipse).getSemiPerimeter(); //Descendant access
  }
}
```

class *Shape* **infers** (keyword for reverse inheritance) two classes which are *Parallelogram* and *Ellipse*. It factorises (keyword **factored**) the attribute *perimeter* which is first defined in these two reverse-inherited classes. The same thing is done for the attribute *area* except that *area* is equivalent to *surface* within class *Ellipse* and needs to be renamed. The attribute *colour* is new and is added to class *shape*. Finally, method *getSemiPerimeter()* uses the descendant access mechanism (keyword **inferior**) in order to access to the feature code defined in class *Ellipse*. Method *getSemiPerimeter()* is now available to class Parallelogram and to all further descendants of class *Shape*

4. Related Works

We make a distinction between **evolution** (or maintenance), where a library of classes is modified by its designer in order to address new needs and, **adaptation** where this library is adapted to a specific use by the designer of a given application.

4.1. *Adaptation of class hierarchies*

Object-oriented database systems which implement dynamic evolution address mainly data intensive applications. [LI 88] describes the "object flavour evolution". The possible evolutions imply that fundamental semantic of objects changes regarding their nature. In [AMA 02] the author proposes mechanisms for flexible and safe adaptations ; it deals with strategy objects and control objects. The former implements adaptation strategies and the latter controls adaptable procedures using the strategy object. About architectural description languages (ADL), [HEI 98a] proposes an approach to adapt Java Beans into new applications. ADL deals with entities such as methods and classes as well as other constructs more specific to the description of components such as ports. Heineman in [HEI 98b] presents a new technique for designing software components and a mechanism for adapting their behaviour. The technique is called *Active Interface* and proposes that software components should have two interfaces one for describing the behaviour and one for adapting it. To alter the behaviour of a component it is necessary to define an intermediate entity which intercepts messages and events. In [DAV 02] the author present two approaches related to application adaptability : *i)* the standardisation of the interfaces and the specification of common behaviour at a very high level in the design, *ii)* the description of a separate component that implements each variation which may be combined on a case-by-case basis when a new system is implemented.

4.2. *Evolution of class hierarchies*

According to the evolution of libraries of classes, an important work has been made by M. M. Lehman about the laws of software evolution [LEH 96]. It is interesting to take a look at the law dealing with continuous change and increasing complexity. Related to this topics, the author says that "*a program that is used must be continually adapted else it becomes progressively less satisfactory*", and that "*as a program is evolved its complexity increases unless work is done to maintain or reduce it*". According to the opinion of [RAJ 02], the evolution of legacy systems is typically dealing with changes in the requirements which has to be fully addressed and understood. He promotes *i)* top-down techniques based on formulation and verification of hypothesis about the code and *ii)* bottom-up techniques based on chunking (chunks are pieces of code with an identifiable meaning : algorithms, functions or data structures). From his point of view it is necessary to much rely on tools in order to help the user. To make evolve a library of classes it is proposed to propagate across all dependencies, the changes or ripple effects using a top-down process. One of the idea that the author promotes is that shortening the propagation chain can be issued using coding and design conventions, constructs with intuitive meaning, comprehensive architectures, documentation. Evolution is also addressed within the context of object-oriented software [CAS 95] ; Every possible modification of a class can be defined in terms of specific atomic update operations. It mentions in particular several approach in order to address evolution : *Tailoring*, *Surgery*, *Versioning* or *Reorganisation*. Tailoring al-

lows the programmer to replace unwanted characteristics from standard classes with properties suitable for new applications. Some of the languages mechanisms used are : renaming, redefinitions, interfaces. Surgery represents all the modifications that have to be applied to classes due to the changes made in the design of the application. Reorganisation is required when important changes are made within the library of classes. Versioning allows to maintain the history of all modifications in a class, during the software life-cycle.

5. Conclusion and Future Work

In this paper we intended to show that introducing reverse inheritance in order to specify changes in a library of classes is an interesting approach. The case-studies that have been presented show the kind of modifications that may be described using reverse inheritance. According to what has been mentioned in section 4 the reader may understand that this approach does not intend to solve all cases of evolution or adaptation. It is more dedicated to changes that do not induce a full rewriting of applications and it does not propose immediate solution for change propagation or versioning. But we promote the idea that thanks to the use of reverse-inheritance controlled by annotations we provide a support for both propagation of changes and versioning. It may be also useful to take into account the possible obsolete features or classes [MEY 02] in order to ensure the compatibility and to warn users about the existence of a better version. Adaptation which is another target of our approach is also addressed by making easier the insertion of new feature or classes, or their redefinitions. It favours both reusability and extendibility of library of classes.

A first short-term issue is to give a more precise definition of the semantics which is behind each mechanisms shown in section 3. In particular, two different cases will be addressed depending on the targeted languages : does it support single or multiple inheritance ? A second one is to propose patterns which describe the transformation of a library including changes (specified with reverse inheritance), into a library of classes implementing only classical inheritance. These patterns should address very carefully the readability of the library after the end of the maintenance process. A mid-term issue is to validate our approach. It will rely on the use of existing parsers dedicated or not to the targeted language. The transformations will be applied to the abstract syntax tree corresponding to the targeted language extended with reverse-inheritance. Its interface could rely on a wizard and be plugged in programming environments such as Eclipse or EiffelStudio.

6. Bibliographie

[AMA 02] AMANO N., WATANABE T., « A Software Model for Flexible and Safe Adaptation of Mobile Code Programs », *International Workshop on Principles of Software Evolution (IWPSE)*, Orlando, Florida, USA, 2002.

[ARN 00] ARNOLD K., GOSLING J., *The Java Programming Language*, Sun Microsystems, 3rd edition, USA, 2000.

[CAS 95] CASAIS E., « Managing Class Evolution in Object-Oriented Systems », *O. Nierstrasz and D. Tsichritzis Object-Oriented Software Composition*, Prentice Hall, 1995.

[CRE 02] CRESCENZO P., LAHIRE P., « Using both Specialization and Generalization in a Programming Language : Why and How ? », *Proceedings of the MASPEGHI Workshop at OOIS 2002*, Montpelier, France, September 2002.

[CRE 03] CRESCENZO P., JALADY C., LAHIRE P., « Annotation of Classes and Inheritance Relationship : an Unifi ed Mechanism in Order to Improve Skills of Libray of Classes », *Proceedings of the MASPEGHI Workshop at ASE 2003*, Montréal, Canada, October 2003.

[DAV 02] DAVIS M. J., « Adaptable, Reusable Code », *ACM-SIGSOFT Symposium on Software Reusability (SSR'95)*, April 2002.

[DIC 96] DICKY H., DONY C., HUCHARD M., LIBOUREL T., « On Automatic Class Insertion with Overloading », *Conference on Object-Oriented*, 1996, p. 251-267.

[GOD 93] GODIN R., MILI H., « Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices », *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, 1993, p. 394-410.

[GOD 02] GODIN R., HUCHARD M., ROUME C., VALTCHEV P., « Inheritance and Automation : Where Are We Now ? », *Proceedings of The Inheritance workshop at ECOOP 2002*, 2002.

[GRO 03] GROUP O. M., « Unifi ed Modelling Language Specifi cation version 1.5, 1st ed. », http ://www.omg.org, March 2003.

[HEI 98a] HEINEMAN G. T., « Adaptation and Software Architecture », *3rd International Software Architecture Workshop (ISAW3)*, Orlando, Florida, USA, 1998.

[HEI 98b] HEINEMAN G. T., « Adaptation and Software Architecture », *22nd Anual International Computer Science and Application Conference (COMPSAC-98)*, Viena, Austria, August 1998, p. 121-127.

[LAH 03] LAHIRE P., « Extensibility and Reusability : A.O.P. vs. O.O.P Case studies in Eiffel and AspectJ », April 2003.

[LAW 94] LAWSON T., HOLLINSHEAD C., QUTAISHAT M., « The Potential for Reverse Type Inheritance in Eiffel », *Technology of Object-Oriented Languages and Systems (TOOLS'94)*, 1994.

[LEH 96] LEHMAN M. M., « Laws of Software Evolution Revisited », *European Workshop Software Process Technology (EWSPT96)*, Nancy, France, October 1996.

[LI 88] LI Q., MCLEOD D., « Object Flavor Evolution in an Object-Oriented System », Contract mda903-81-c-0335, 1988, Defense Advanced Research Projects Agency.

[LIE 95] LIEBERHERR K., « Workshop on Adaptable and Adaptive Software », *Addendum to the Proceedings of OOPSLA 95*, 1995.

[MAR 00] MARTIN R., « Design Principles and Patterns », http ://www.objectmentor.com, 2000.

[MEY 02] MEYER B., « Eiffel : The Language », http ://www.inf.ethz.ch/ meyer/, September 2002.

[RAJ 02] RAJLICH V., « Comprehension and Evolution of Legacy Software », *International Conference of Software Engineering (ICSE97)*, Boston, USA, 2002.

[SAK 02] SAKKINEN M., « Exheritance - Class Generalization Revived », *In European Conference on Object-Oriented Programming*, 2002.