Reverse Inheritance : Improving Class Library Reuse in Eiffel

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

[*] *University "Politehnica" of Timişoara*
Faculty of Automatics and Computer Science
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.

ABSTRACT. *Reusing Eiffel class libraries can be problematic. Modifying the existing source code of classes, if is available and is not copyrighted, involves maintaining the entire class library. Reverse inheritance class relationship offers several facilities for reorganizing Eiffel class hierarchies: creating a common superclass, factoring common features, inserting a class into an existing hierarchy. Using a new class relationship similar to ordinary inheritance with a symmetrical semantics will guarantee the expresiveness of the new class design without changing the original source code of the reused classes.*

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

KEYWORDS: *reverse inheritance, reuse, adaptation, evolution, hierarchy reorganization, object-oriented programming languages*

1re soumission à *LMO*, le 7 novembre 2006.

1. Introduction

Developping reusable components and frameworks [GAM 97] in object technology is hard [OPD 92]. In this paper we will present a new class relationship, named reverse inheritance (RI), designed as a language extension, which makes easier the reuse of Eiffel [MEY 02] class libraries, adaptation of existing class hierarchies for a specific context and restricted evolution of hierarchies. The choice for Eiffel is motivated in section 3. Modifying the source code of class libraries implies later maintainance of the whole library. The problem is even more severe when the source code is copyrighted or even worse, when it is unavailable. Reverse inheritance combined with ordinary inheritance can help in class hierarchy reorganization as it will be shown in section 2.
The idea of upward inheritance was born in the database world from the concept of database schema generalization [SCH 88]. A type corresponding to a database schema may be a generalization of several specialized ones. It is also the case of generalization in a global multi database view which provides a homogeneous interface to a set of heterogeneous databases. The basic idea of reverse inheritance class relationship is the generalization abstraction [SMI 77], which enables a set of individual objects to be thought generically as a single named object. It is considered to be one of the most important mechanism for conceptualizing the real world : generalization helps the goal of uniform treatment for objects of the models which are obtained. From the development point of view of a software system, direct inheritance is a top-down approach of construction while reverse inheritance offers the possibility of constructing software in a bottom-up manner. We adhere to the idea that it is more natural to first create the subclasses, then to observe and analyze commonalities, and after that to define the superclasses [PED 89, SAK 02]. The autonomous design of class hierarchies or database schema will give rise to inhomogeneities. Their reusability depends strongly on their capabilities of adapting their local interface to a common global interface.
The reverse inheritance class relationship is also known as *exheritance* [SAK 02], *adoption* [LAW 94], *generalization* [PED 89, UML04] or *upward inheritance* [SCH 88].
The source class of reverse inheritance is known as generalizing class [SAK 02] or as foster class [LAW 94]. In the state of the art there are several approaches dealing with reverse inheritance issues in domains like object-oriented programming and design, databases, artificial intelligence. We start from the definition of reverse inheritance given by Pedersen [PED 89, SAK 02] which states that a class G can be defined as a generalization of A_1, A_2, . . ., A_n previously defined classes. If the value of n is 1 then we discuss about single generalization, otherwise about multiple generalization. Informally it can be defined as another model of inheritance where the subclass exists and the superclass is constructed afterwards.

The paper is structured as follows. In section two we will present how reverse inheritance can handle several use cases of class hierarchy reuse. Section three discusses semantical elements of reverse inheritance. Section four presents some dynamic binding aspects in the context of reverse inheritance. In section five are presented ideas about the feasibility of the approach and how the semantics of the reused system can

be translated in an equivalent compilable source code. In section six related works are presented and comparisons to other reuse mechanisms are discussed. In section seven conclusions are drawn and future works are stated.

2. Reverse Inheritance Based Solutions

In this section we will present several situations in which reverse inheritance can bring a substantial contribution to class library reuse.

2.1. *Capturing Common Functionalities*

In this use case we intend to address the situations in which we need to use in a uniform manner several classes which come possibly from different class hierarchies and have common operations. This will imply setting a new superclass for all the selected subclasses. One possible solution is to use ordinary inheritance and to modify the source code of the class hierarchies accordingly, in order to obtain the desired class configuration. All the subclasses must be altered because the new superclass must be listed. This approach involves maintaining later on all the libraries from which the subclasses came from.

A different approach based on reverse inheritance will overcame this problem. By creating a superclass using reverse inheritance there is no need to modify the source of subclasses, because in the new superclass can be specified a list of all its subclasses. In figure 1 we have such an example where classes *RECTANGLE* and *ELLIPSE* come from different libraries but share a common feature *draw()*. Using ordinary inheritance a new class *TRIANGLE* was created as a subclass of *SHAPE*, thus two class hierarchies can be combined. The example in figure 1 is a simplified and an optimistic

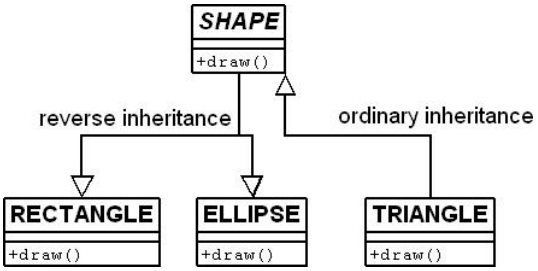

Figure 1. *Capturing Common Functionalities*

one. In practice there should be addressed problems like name conflicts and signature incompatibilities. Name conflicts [PED 89, LAW 94] arise when two features having the same semantics have different names - called lost friends [SAK 02] and when two features having the same name but different semantics - called false friends. Signature

incompatibilities have to take into account parameter and return types, parameter number and order, assertions : preconditions, postconditions and invariants. These issues will be addressed in section 3.

The benefits of the approach in this case is that we do not have to modify the class hierarchy, we factored the common features into the superclass, we created a common interface which helps manipulating the subclasses in an uniform manner and we extended the hierarchy with a new subclass.

2.2. *Inserting a Class Into an Existing Hierarchy*

Another use case we address, refers to modifying class hierarchies which were designed quickly and the result is not general enough. Also we address situations in which unforseen changes have to be applied to an existing class hierarchy.

In figure 2 is presented a typical situation in which the design of an existing class hierarchy have to be changed and a new class have to be inserted between already existing two ones. To add retroactively a new layer of abstraction in a class hierarchy is a natural practice when the model of an application has to be adapted to new contexts or when the model evolves.

In the initial class hierarchy existed two classes : superclass *SHAPE* and subclass *RECTANGLE*. Later a new class *PARALLELOGRAM* is needed which is more general than class *RECTANGLE* is and it should be also a subclass of *SHAPE*. There are several possibilities to reorganize the class hierarchy using ordinary inheritance, but this implies modifications and thus maintenance of the original source code. The

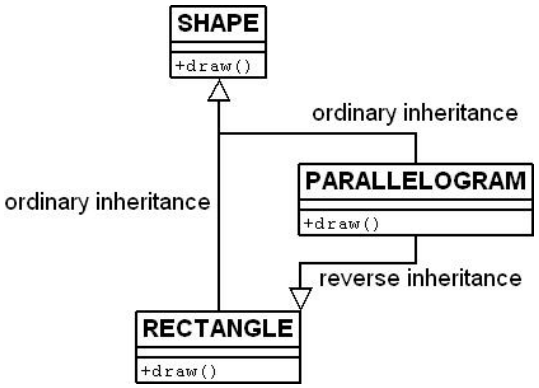

Figure 2. *Inserting a Class Into an Existing Hierarchy*

reverse inheritance solution implies creating class *PARALLELOGRAM* as a subclass of *SHAPE* by ordinary inheritance and also as a superclass of *RECTANGLE* by reverse inheritance class relationship. Public and protected features are inherited from

SHAPE superclass into *PARALLELOGRAM* class by ordinary inheritance and from *PARALLELOGRAM* into *RECTANGLE* by reverse inheritance.

The gain of this method is preserving the original classes unmodified and still refining the old class hierarchy. Because the source code of the original classes is not altered, it is very easy to cancel the modifications.

3. Expressiveness of Exheritance

In this section we will present the main semantical elements of reverse inheritance which helps class hierarchy reuse. First of all we have to choose a programming language in which reverse inheritance can be naturally integrated. Several statical object-oriented languages were considered : Java, C++ and Eiffel. When adding an extension to a language the philosophy of the language should be respected and unnatural constructs should not be allowed. According to this and because of the symmetry which exists between ordinary and reverse inheritance, the latter should not introduce class constructs which may not be built using ordinary inheritance only. In Java because there is no multiple inheritance between classes only between interfaces, the symmetry principle requires reverse inheritance to be allowed only for interfaces. In C++ and Eiffel multiple ordinary inheritance will enable, symmetrically, multiple reverse inheritance. Both Java and C++ do not have adaptation mechanisms for the features in the sense Eiffel has renaming and covariant redefinition. C++ and Java offer only non-variant redefinition for the implementation of methods. In Java and C++ function overloading is allowed while in Eiffel each feature requires to have a unique name. In Eiffel, features implemented by memory (attributes) or computation (methods) are used in an uniform manner. The choice for Eiffel programming language was taken because from our point of view the features that it provides and the cleanness of the approach according to the points mentioned above makes possible the definition of an expressive and orthogonal reverse inheritance relationship. In order to respect the duality of ordinary inheritance of Eiffel, reverse inheritance can be conforming or non-conforming. Conforming inheritance/reverse inheritance keeps the type conformance relationship between subclass and superclass while non-conforming does not.

3.1. *Factoring Features, Exheriting Implementation and Renaming*

When there are several features with the same semantics in the subclasses it is natural to expect one, deferred or effective in the foster class. Implicitely the attributes are factored as attributes in the foster class, while methods are better to be exherited as deferred features. Feature factorization can be implicit or explicit using the keywords *ALL EXCEPT feature_list* respectively *NONE EXCEPT feature_list*. In the first case all features with compatible signatures are exherited automatically in the foster class, exceptions can be listed if needed. In the second case only the listed features are

exherited. Implementation of features can be selected from one of the subclasses if all the dependent features are exherited [SAK 02] or redefined at the foster class level.

```
class RECTANGLE
 feature
   forecolor, bgcolor : INTEGER;
   draw is do ... end
   floodfill is do ... end
   perimeter : INTEGER is do ... end
   print(xcenter, ycenter : INTEGER) is do ... end
   display is
    require forecolor > 64 and bgcolor < 192
   do
     ...
   end
end - class RECTANGLE

class ELLIPSE
 feature
   forecolor,bgcolor : INTEGER;
   draw is do ... end
   floodfill is do ... end
   circumference : INTEGER is do ... end
   print(x1, y1, x2, y2, color : INTEGER) is do ... end
   display is
    require forecolor <> bgcolor
   do
     ...
   end
end - class ELLIPSE

foster class SHAPE
 exherit
   RECTANGLE ALL
    undefine draw
    undefine floodfill
    rename perimeter as boundary
    undefine boundary
    adapt print
   end
   ELLIPSE ALL
    undefine draw
    move floodfill
    rename circumference as boundary
    undefine boundary
    adapt print
   end
 feature
   print(x1, y1, x2, y2 : INTEGER) is
    require stronger
   adapted
    RECTANGLE.print((x1+x2)/2,(y1+y2)/2)
    ELLIPSE.print(x1,y1,x2,y2,0)
   do
    - possible implementation
   end
end - class SHAPE
```

Figure 3. *An illustration of the extension of reverse inheritance in Eiffel*

In the example proposed in figure 3, classes *RECTANGLE* and *ELLIPSE* exist originally and foster class *SHAPE* is created afterwards by reverse inheritance. Feature

draw is factored as a deferred feature in the foster class. One may notice that it is undefined on both reverse inheritance branches (these two clauses could be avoided because it is the default) . Feature *floodfill* is factored and the implementation in the foster class *SHAPE* is taken from class *ELLIPSE*, there is one clause *undefine* on the exheritance branch coresponding to class *RECTANGLE* and one clause *move* on the exheritance branch corresponding to class *ELLIPSE*. From this point of view reverse inheritance is symmetrical with multiple inheritance when selecting the implementation for a multiply inherited/exherited feature. We assume also that all conditions regarding implementation exheritance are respected. Renaming is performed on feature *perimeter* from class *RECTANGLE* and *circumference* from class *ELLIPSE*, so that class *SHAPE* provides one corresponding deferred feature *boundary*.

Several restrictions regarding the foster class have to be taken into account. In order to keep the behavior of the subclasses unchanged, it is not allowed to add new features (attributes and methods) in the superclass ; only the common ones can be factored as deferred (abstract) from the subclasses. At the implementation level, when an executable has to be built, some modifications in the code, if available, are possible, still keeping the behavior unchanged.

3.2. *Parameter Adaptations*

Because signatures of semantically equivalent features are not always compatible we can use the renaming mechanism discussed in the previous section to get a common name. We think that it is not enough and some adaptations have to be performed on parameters also. In [LAW 94] a simple and straightforward syntax is presented for parameter reordering. The example of figure 3 presents a way for adapting features with not the same number of parameters. In foster class *SHAPE* a feature *print* has four parameters of type *INTEGER* ; it is factored from class *RECTANGLE* and *ELLIPSE* where it has respectively two and five parameters. It is necessary to adapt the signature in order to handle the case where an instance of *RECTANGLE* or *ELLIPSE* is used through a reference of type *SHAPE*. When calling feature *print* from *RECTANGLE* its two parameters must be computed from the four parameters provided by the signature of the foster class. In the same way, calling feature *print* from *ELLIPSE* needs to fill five parameters so one more than *print* from foster class.

3.3. *Assertion Adaptations*

In [LAW 94] the authors analyse the problems of assertion adaptation. The precondition in the foster class is proposed to be created by applying the logical *AND* operator on the preconditions in the subclasses, respectively the postcondition is obtained by applying the *OR* logical operator on the postconditions and invariants of the subclasses. In practice not all features are factored, so assertions containing them, can be adapted by replacing the missing features with the neutral values of the logical operators. To guarantee statically that the precondition in the foster class is stronger

than all the preconditions in the subclasses is a NP-complete problem, so keywords are used for making the programmer aware of his responsablity for defining them correctly. Possibly, he may redefine the assertion ; he ensures implicitly that the assertion satisfies the constraint infered by the keyword *weaker* or *stronger* depending on the type of assertion. For postconditions the problem is in theory not so severe because a postcondition can be always replaced with *true*, which is the weakest postcondition. Anyway, if there is no suitable assertion then for safety reason non-conforming reverse inheritance should be chosen.

In example of figure 3, feature *display* in both subclasses is equipped with preconditions. The two features involved in the assertions *forecolor* and *bgcolor* are implicitely exherited from the subclasses into the foster class with the same names. The precondition in the foster class is built from the two preconditions defined in the subclasses using the *AND* logical operator and the *require stronger* keywords denoting that the consistency of the precondition is under the responsability of the programmer. For postconditions this is *ensure weaker* that would be used.

4. Exheritance from the Runtime Point of View

4.1. *Dynamic Binding and Exheritance Clause Combinations*

In the example of figure 2 the type conformance between *PARALLELOGRAM* and *SHAPE* respectively *RECTANGLE* and *PARALLELOGRAM* is achieved by the combination of ordinary and reverse inheritance. Thanks to it, the same feature *draw* is available in all the classes of the hierarchy.

Depending on the clause combinations the implementation of *draw* in *PARALLELOGRAM* may come from one of the three classes. Let us study some interesting needs :

– The suitable implementation of feature *draw* is the one of class *SHAPE* ; we suppose that it is effective. It is obtained implicitly by the ordinary inheritance relationship set between *PARALLELOGRAM* and *SHAPE*. The version from class *RECTANGLE* must be undefined if it is also effective.

– Class *PARALLELOGRAM* needs to own the implementation of *draw*, then it is necessary to redefine the one of class *SHAPE* whereas the version from class *RECTANGLE* has to be undefined.

– The suitable implementation is the one of class *RECTANGLE*. It is obtained by moving (through the reverse inheritance relationship set between *RECTANGLE* and *PARALLELOGRAM*), the implementation of *draw* in *PARALLELOGRAM*. The version from *SHAPE* must be undefined or redefined if is effective.

– Finally, if the feature *draw* is expected to be deferred then both implementations (from classes *SHAPE* and *RECTANGLE*) have to be undefined.

Figure 4, shows how polymorphism works on foster class : an attribute of type *PARALLELOGRAM* may receive an object of type *RECTANGLE* but the reverse is false and an attribute of type *SHAPE* may receive an object of type *PARALLELOGRAM* or *RECTANGLE*. The version of feature *draw* which is used using dynamic binding depends on the cases presented above. Please note that for space reason we did not address repeated inheritance (see next section for some details).

```
...
 sref : SHAPE ;
 pref, pobj : PARALLELOGRAM ;
 robj : RECTANGLE ;
 ...
  create pobj ; create robj ;
  - version of draw depends on the clause combination
  pref = robj ; pref.draw
  sref = robj ; sref.draw
 ...
```

Figure 4. *Possible use of classes of figure 2 in a client class*

4.2. *Some Problems to be Addressed*

The body of feature *draw* in class *RECTANGLE* may contain the keyword *precursor* (its semantics is close form the keyword *super* of Java). This needs to be taken into account according to the cases mentioned above. If the implementation comes from class *SHAPE*, meaning that feature *draw* is undefined on the reverse inheritance branch, then the precursor is the same as without reverse inheritance and that is correct. If the implementation of the feature is exherited from *RECTANGLE* into class *PARALLELOGRAM* by undefining the version inherited from *SHAPE*, the precursor will call the aproppriate *draw* version from class *SHAPE*. But, if the feature has a new implementation in class *PARALLELOGRAM* the behavior of the subclass is changed, so this kind of situations must not be allowed. Moreover when feature *draw* is undefined in the foster class, the subclass will have no precursor implementation available, so this situation has also to be forbidden.

When building diamond class configuration with class *A* as root base class, subclasses *B* and *C* as descendants of *A*, and class *D* as a descendant of *B* and *C* it is possible to use ordinary and reverse inheritance and to insert the classes in the hierarchy in different orders. When sharing repeatedly inherited features through different inheritance paths created by ordinary and reverse inheritance there is no dynamic binding problem, because there is one copy of the feature. When features are replicated and class *A* is created last using reverse inheritance the selection of the feature for class *D* must be specified in class *A*, like "select f_final in D". When class *D* is created last, any combination of reverse and/or ordinary inheritance between the rest of the classes, designed for replication, will use the classic selection mechanism of Eiffel.

5. From Inheritance/Exheritance Hierarchies to Pure Inheritance

In order to point that our approach is feasible we will show that each semantical construct discussed earlier can be expressed using a pure Eiffel language. The intermediate compilable code may contain a modified copy of the original source code. Modifications are mostly performed at syntactical level, leaving the behavior unchanged.

In example of figure 5, we present how feature related to reverse inheritance detailed in the example of figure 3, can be translated into equivalent Eiffel code.

First we must mention that deferred class *SHAPE* is now superclass of both *RECTANGLE* and *ELLIPSE* subclasses. The feature *draw* is deferred in the superclass and effective in the subclasses. The implementation of feature *floodfill* is copied from subclass *ELLIPSE*.

In this example we present also how parameter adaptation mechanism can be implemented. The main ideea proposed is to rename the original feature *display* in each subclass (copy of the original subclass source code) and then to add the new feature *print* with four parameters. In each subclass the new feature will contain an adapted delegation call to the original feature.

The precondition of feature *display* in superclass *SHAPE* is implemented using the *AND* logical operator on the corresponding preconditions in the subclasses.

Any diamond class combination created by ordinary and reverse inheritance can be implemented using just ordinary inheritance. In the case of feature replication, the select clause from the foster class must be moved into the last subclass.

In example of figure 6 we show how the class hierarchy defined in figure 2 can be implemented in pure Eiffel language and that the two class hierarchies are equivalent.

It can be noticed that the three classes are put in the same hierarchy in their natural order. Feature *draw* in class*PARELLELOGRAM* is redefined and has its own implementation. The new implementation will not affect the behavior of the subclass because in *RECTANGLE* there is a different version available. Type conformance between classes of the original class hierarchy still holds in the equivalent implementation.

6. Related Works

In this section we analyze several works related to class hierarchy reorganization, class reuse mechanisms, software adaptation and evolution.

In [GAM 97] are presented several design patterns which are a collection of general solutions to commonly occuring problems. The class reorganization schemes are either applied at the design time or they are applied afterwards, but this requires changing the original code.

```
deferred class SHAPE
 feature
  forecolor, bgcolor : INTEGER;
  draw is deferred ... end
  floodfill is
  do
   - copy implementation of feature floodfill from class Ellipse
  end
  boundary is deferred ... end
  print(x1, y1, x2, y2, color : INTEGER) is do ... end
  display is
   require forecolor > 64 and bgcolor < 192 forecolor <> bgcolor
  do
   ...
  end
end - class SHAPE

class RECTANGLE
 inherit
  SHAPE
   rename boundary as perimeter
   rename print as old_print
  end
 feature
  draw is do ... end
  floodfill is do ... end
  perimeter : INTEGER is do ... end
  old_print(xcenter, ycenter : INTEGER) is do ... end
  print(x1, y1, x2, y2 : INTEGER) is
  do
   old_print((x1+x2)/2, (y1+y2)/2)
  end
  display is
   require else forecolor > 64 and bgcolor < 192
  do
   ...
  end
end - class RECTANGLE

class ELLIPSE
 inherit
  SHAPE
   rename boundary as circumference
   rename print as old_print
  end
 feature
  draw is do ... end
  floodfill is do ... end
  circumference : INTEGER is do ... end
  old_print(x1, y1, x2, y2, color : INTEGER) is do ... end
  print(x1, y1, x2, y2 : INTEGER) is
  do
   old_print(x1, y1, x2, y2, 0)
  end
  display is
   require else forecolor <> bgcolor
  do
   ...
  end
end - class RECTANGLE
```

Figure 5. *Example of figure 3 using ordinary inheritance only*

```
class SHAPE
 feature
   draw is do ... end
end - class SHAPE

class PARALLELOGRAM
 inherit
   SHAPE
     redefine draw
   end
 feature
   draw is do ... end
end - class PARALLELOGRAM

class RECTANGLE
 inherit
   PARALLELOGRAM
     redefine draw
   end
 feature
   draw is do ... end
end - class RECTANGLE
```

Figure 6. *Example of figure 2 using ordinary inheritance only*

In [OPD 93] is described a manual method of reorganizing class hierarchies by creating a new abstract superclass for a set of subclasses. It is explained step by step the process of creating an abstract superclass : adding function signatures to the superclass, making function bodies compatible, moving variables and migrating common code to the superclass. In our work dedicated to Eiffel we encapsuled all these operations in the semantics of reverse inheritance.

In [FOW 99] are presented several techniques of restructuring code by altering its internal structure without changing external behavior. This philosophy was used in our work when we expressed the semantics of reverse inheritance in terms of equivalent pure language constructs.

In [DAO 02] is presented an algorithm that reorganizes class hierarchies based on Galois lattice for optimizing factorization of features. In this work the changes are intended to avoid the flaws regarding factorization. Modifications of attributes to all occurences is considered time consuming and error prone. Moreover, multiple unnecessary declarations of features makes the hierarchy less understandable and usable.

In [SCH 02, SCH 03] is presented the trait model which can be viewed as a class reusing mechanism because traits are reusable and composable parts of a class. This approach can be applied only if traits are already defined while reverse inheritance can be applied to any class hierarchy written in Eiffel.

7. Conclusion and Future Work

We showed that reverse inheritance can help Eiffel class reusability by redesigning existing class hierarchies. We thought about the main problems and we brought some solutions. The new class relationship is built symetrically from ordinary inheritance and is equipped with a specific adaptation mechanisms, which does not represent a severe deviation from the philosophy of the language. The semantical elements were analysed from two perspectives : static behavior (definition, adaptations) and dynamic behavior (dynamic binding). There were provided two examples trying to increase the interest of using reverse inheritance. We showed also that the approach is feasible by providing compilable code with equivalent semantics.

One of the perspectives regarding the semantics of reverse inheritance is to fully integrate it into Eiffel programming language. In order to achieve this goal, a formal model for the foster class must be proposed. This will allow designers to use foster classes in their hierarchies. Next, a translator must be built in order to generate compilable Eiffel code. The integration of the translator into an industrial development environment like Eclipse would automate the class library reuse.

8. Bibliographie

[DAO 02] DAO M., HUCHARD M., LIBOUREL T., ROUME C., « Evaluating and Optimizing Factorization in Inheritance Hierarchies », *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.

[FOW 99] FOWLER M., *Refactoring Second Edition*, Addison-Wesley, 1999.

[GAM 97] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1997.

[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.

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

[OPD 92] OPDYKE W., « Refactoring Object-Oriented Frameworks », PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[OPD 93] OPDYKE W. F., JOHNSON R. E., « Creating Abstract Superclasses by Refactoring », 1993.

[PED 89] PEDERSEN C. H., « Extending ordinary inheritance schemes to include generalization », *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1989, p. 407–417.

[SAK 02] SAKKINEN M., « Exheritance - Class Generalization Revived », *Proceedings of the Inheritance Workshop at ECOOP*, Malaga, Spain, June 2002.

[SCH 88] SCHREFL M., NEUHOLD E. J., « Object Class Definition by Generalization Using Upward Inheritance », *IEEE Transactions*, 1988.

[SCH 02] SCHÄRLI N., DUCASSE S., NIERSTRASZ O., « Classes = Traits + States + Glue (Beyond mixins and multiple inheritance) », *Proceedings of the International Workshop on Inheritance*, Malaga, Spain, June 2002.

[SCH 03] SCHÄRLI N., DUCASSE S., NIERSTRASZ O., BLACK A., « Traits : Composable Units of Behavior », *Proceedings of the Inheritance Workshop at ECOOP 2003*, Darmstadt, Germany, July 2003.

[SMI 77] SMITH J. M., SMITH D. C., « Database Abstractions : Aggregation and Generalization », *ACM Transactions on Database Systems*, vol. 2, June 1977, p. 105–133.

[UML04] « UML Superstructure Version 2.0 », www.omg.org/uml, October 2004.