Buletinul Stiintific al Universitatii "Politehnica" din Timisoara, ROMANIA
Seria AUTOMATICA si CALCULATOARE
PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE
Vol.49 (63), 2004, ISSN 1224-600X

Factoring Mechanism of Reverse Inheritance

Ciprian-Bogdan Chirila*, Pierre Crescenzo**, Philippe Lahire**,
Dan Pescaru*, Emanuel Tundrea*

* Department of Computer Science, Automation and Computer Science Faculty,
University "Politehnica" of Timisoara, V. Pârvan no. 2, Timisoara, Romania
chirila@cs.utt.ro, dan@cs.utt.ro, emanuel@emanuel.ro

** I3S Laboratory (UNSA/CNRS), University "Sophia Antipolis" Nice, Les Algoritmes,
bat. Euclide B 2000, Route des Lucioles BP121,F-06903 Sophia Antipolis CEDEX, France
Philippe.Lahire@unice.fr, Pierre.Crescenzo@nom.fr

Abstract ***– In this paper we present a new approach for facilitating the maintenance, reengineering and adaptation of class libraries designed using object-oriented technology. The technique uses a new class relationship called reverse inheritance. We strive to prove that using this class relationship with it's factoring supporting mechanism it is possible to factor features from a hierarchy, to add new features to a hierarchy, and to connect two class hierarchies. Also in the paper a list of problems relative to the new approach is formulated.***

Keywords: ***inheritance relationship, reverse inheritance relationship, reengineering.***

I. INTRODUCTION

Inheritance is the most difficult and the most problematic issue in the object-oriented programming. This feature distinguishes object-oriented programming from object-based programming or from other modern programming languages. Inheritance is defined informally as [5]: an incremental modification mechanism that transforms the ancestor class into a descendent class by augmenting it in a various way. The informal definition is broadly accepted by researchers, but it's implementations are very different. Another definition is given in [17], inheritance is viewed as a shorthand mechanism for defining a new class relative to an existing class specifying only the differences, inheriting all the existing features. As many implementations for inheritance exist, as many object-oriented paradigms are to be considered [5].

In this paper we present the model of a new kind of inheritance class relationship and we show the motivations behind the supporting mechanisms that are needed. Direct inheritance models are downward type of inheritance, while what we propose, reverse inheritance for software adaptation and evolution, is an upward type of inheritance. It is more natural to define concrete subclasses and then to extract commonalities into superclasses [11].

In [3] are presented the **main use cases** of reverse inheritance. The need for reverse inheritance is motivated because of the many facilities offered: i) to share common functionalities; ii) inserting a class into an existing hierarchy; iii) extending an existing hierarchy; iv) adding features to a class; v) factoring features from classes. The proposed technique is applied to class libraries and components - in the sense of hierarchies of classes stated by Bertrand Meyer in [8].

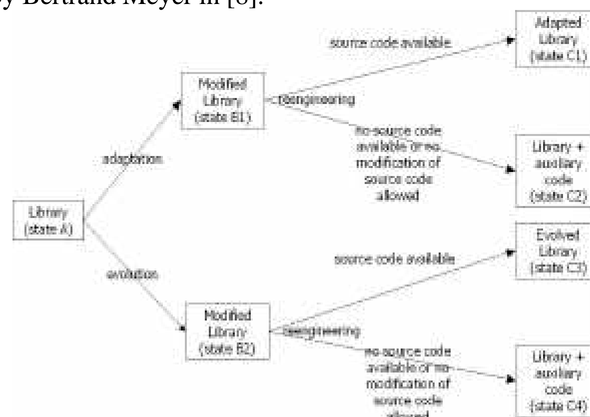

Fig. 1. Approach

The diagram from figure 1 depicts the states through which the library is transformed in order to reach evolution and adaptation goals. We start from state A, the original state of the library that we want to reengineer. The choice is dichotomist, the available possibilities are: adaptation or evolution. By adaptation we mean changes that will make the classes suitable to a particular context: removing unused features, adding new ones, renaming existing ones. By evolution we refer to changes that keep the original interfaces intact in order to be compatible with older clients. Then another two cases appear for each choice: wheatear we have or not access to modify the source code. Because of many reasons source code may not be modified: i) software may be copyrighted; ii) software have to be unchanged for existing applications; iii) software is maintained by third party; iv) software of source code is not available e.g. precompiled libraries. In state B (B1 or B2) the hierarchies are linked by inheritance and reverse

inheritance. In this state are applied the changes towards a new context or a new purpose of the classes. State C is the final one, in which only direct inheritance class relationship is used, all the reverse inheritance link being transformed automatically into native language class relationships. The set of code transformations we are using deals with signature adaptation and uses the following basic actions: i) method / parameter renaming; ii) parameter addition / removal / reordering; iii) method parameter type changing. Now the library code is human readable, refactored, adapted, evolved, ready to be integrated into the new context.

The structure of the paper is the following: In the first section it is presented the new class relationship relative to it's symmetrical, normal or direct inheritance. In section two we present our approach relative to reverse inheritance and possible use cases. In the third section factoring mechanism it is described. The fourth section talks about the motivation for the renaming mechanism and related problems. In the fifth section related works are exposed with their differences from our work. In the sixth section conclusions are extracted and future works are traced.

II. ABOUT FACTORING SEMANTICS

A. Inheritance Semantics

We strive to define reverse inheritance relative to the definition of normal inheritance and to assign semantics to it. A first possible informal definition follows: *Reverse inheritance is defined as the inheritance's symmetrical relationship*. At the code level we, introduce two new keywords *infers* which intends to be the symmetrical of *extends*, and *foster* [7] by which the base class of reverse inheritance is denoted:

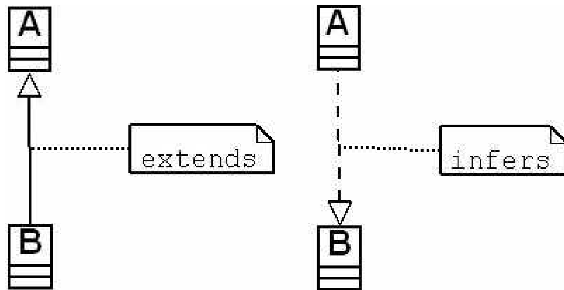

Fig. 2. Infers

```
// used by designer
class B extends A {}
// used by programmer, maintainer
foster class A infers B {}
```

This definition is very ambiguous and a new one, more formal is needed. So the best way in finding a definition, is to start from one of the formal definitions of inheritance, from literature [14, 15, 16, 17, 19]. The selected one is the following: derived = base ⊕ extra, where the union with override operator ⊕ is defined as:

$$\forall \boldsymbol{a},\boldsymbol{b},\boldsymbol{g}.\oplus : (\boldsymbol{a} \to \boldsymbol{b})\times(\boldsymbol{a} \to \boldsymbol{g}) \to (\boldsymbol{a} \to \boldsymbol{b} \cup \boldsymbol{g})$$

$$\oplus = \boldsymbol{l}(f : \boldsymbol{a} \to \boldsymbol{b}).\boldsymbol{l}(g : \boldsymbol{a} \to \boldsymbol{g}).$$

$$\{k \mapsto v \mid (k \in dom(f) \cup dom(g)) \wedge$$

$$(k \in dom(g) \Rightarrow v = g(k)) \wedge (k \notin dom(g) \Rightarrow v = f(k))\}$$

The union with override operator is a standard mathematical operator that combines two maps in a special way, which is defined above. A map is a collection of name-value pairs called maplets (e.g. {a→x, b→y, …}). Objects are modeled as records or maps where each field consists of a label that maps to a value. Operator ⊕ takes two maps and creates a new map that contains all the maplets that satisfy the given condition. The domain values k are taken from the reunion of both argument map domains. The range values v are obtained in the following way: if k is in the domain of the second argument g than it's value is used, otherwise the value of the first argument f it is used. In the case of conflicting labels are preferred the values from the second map argument g.

B. Reverse Inheritance Semantics

In the context of the inheritance model presented, we define our reverse inheritance class relationship and it's associated mechanisms in a very similar way:

$$\text{base} = \text{factored} \oplus \text{extra}$$

$$\text{factored} = \text{derived}_1 \cap \text{derived}_2 \cap \ldots \cap \text{derived}_m$$

We must remind the context of our approach that the base class also called *foster* [7] has to be created and managed by the programmer or the maintainer, and that the rest of the derived classes, that will be adapted may belong to different libraries, class hierarchies.

The first line defines the *base* class of a reverse inheritance which consists of two component records *factored* and *extra*. The main idea is: *to extract common features from library classes an to put them into the foster class*. We consider common features all the features that [7]: i) have the same name in each class; ii) exists a signature that conforms to the signature of each feature in each class. To be more specific, fields should be treated differently from methods in the following way: i) moving the fields to foster class; ii) creating abstract methods in foster class corresponding to each factored item from the derived classes. The motivation behind this idea is to avoid data duplication and to benefit from the dynamic linking of polymorphism when calling methods using foster class references. The control of the code, data and behavior is better if it is centralized in the foster class. As described in the use cases of [3] code using reverse inheritance class relationship flexibility is achieved (e.g design patterns like Composite [6] can be more easily applied).

The factored component record contains all the common features from the derived classes (derived_i). This idea refers the factoring use case enumerated in section I. It's content is denoted by the intersection of all features from the derived classes. All the common features extracted from the derived classes are called factored features. There is also the possibility to add new features - grouped in a

record named extra - to the library classes. With this object record we address the feature adding use case described in section I.

C. Intuitive Approach Through an Example

In this subsection through an example it is presented how the proposed reengineering process works. It is emphasized at each level the operations that adapt library classes.

a) State A: In our practical sample, we start from two library classes Rectangle and Ellipse (state A, on the schema 1) that belong to two different hierarchies and are planned to be used together in a new context:

```
class Rectangle {
  double perimeter;
  double getArea()
  {/*rectangle implementation*/}}
class Ellipse {
  double perimeter;
  double getArea()
  {/*ellipse implementation*/}}
```

Analyzing the code we can conclude that perimeter member and getArea() method can be factored. An automated technique in matching signatures is presented in [20], defining a method of organizing, navigating through, retrieving from software libraries. Signature matching is the process of determining which library components "match" a query signature. There are considered cases of exact matching and various flavors of relaxed matches, based on type built-in information.

b) State B: In state B of figure 1 the reengineering process is initiated and a new foster class is created containing the factored features. We will demonstrate that using the foster class it is possible to manage collection of shapes (e.g to compute the area of the collection uniformly, to modify the perimeter of each item, and even to add a new field: color).

```
foster class Shape infers Rectangle,
Ellipse {
 factored double perimeter;
 /*no implementation, abstract method*/
 factored double getArea();
 /*extra attribute*/
 int color;}
```

In the code above, it is presented the structure of the new foster class, it contains two factored features: perimeter and getArea() and one extra feature color. The factored features were treated differently because one is attribute and the other is method. The former was moved from their original location classes Rectangle and Ellipse to the foster class Shape. The letter, getArea() being a method it had to be abstracted through the abstract method factored double getArea() inserted into the foster class. Methods and attributes have to be treated differently because attributes with the same signature give the same state component in each object, while methods with same signatures may have different implementations in different classes.

c) State C: In state C some modifications at the structure of derived classes were made, eliminating the duplicated data (a copy was in the derived class and one in the foster class):

```
class Rectangle {
  double getArea()
  {/*rectangle implementation*/}}
class Ellipse {
  double getArea()
  {/*ellipse implementation*/}}
```

After all the reengineering process, the transformations that were made, make possible code constructs like:

```
Shape s1=new Rectangle();
Shape s2=new Ellipse();
...
s1.getArea();
s2.getArea();
```

As seen in the code above, in the new context, collection of shapes can be very easily managed by the foster class, not depending on the type of the instance: s1 and s2 are of type Shape even they reference objects of types Rectangle and Ellipse. This is possible due to the transformations made with the help of reverse inheritance and polymorphism. If a Composite or Strategy design patterns [6] are necessary, in this state they can be achieved without making any modifications to shape classes. This sample presents an ideal situation, in which the factored features have the same signature as in subclasses. In real situations signatures of factored features are not identical and they have to be adapted using code transformations pointed out in section I.

III. PROBLEMS TO ADDRESS

In this section will tackle about the problems and restrictions that may arise in applying our adaptation and evolution technique. Also there will be pointed awkward situations that may appear when inheriting classes (combinations of direct and reverse) and motivated the acceptance decisions.

A. Name Conflicts

The factoring mechanisms presented in 2 may not be used in several situations because of name conflicts. Not always features with the same semantics have the same names, because they were developed independently in different class hierarchies. To perform factoring, name conflicts have to be eliminated, renaming being one possible solution. The capability to rename a feature during inheritance is found in programming languages such as Eiffel, C++, Java, Smalltalk: i) in Eiffel methods are allowed to be renamed locally on demand [9]; ii) in C++ global names are used when combining methods [18, 13]; iii) in Java and Smalltalk a pseudo-variable *super* allows the programmer to choose between methods with duplicated names [1]. So renaming could be used in order to: i) make available to possible client-classes, a name which is more suitable to the local context; ii) handle possible name conflicts: two features having the same name may not address always the same functionality. In [12] Pedersen distinguishes two kinds of name conflicts in multiple generalization between two names N1 and N2 from two subclasses A1 and A2. The first conflict arises

when the names N1 and N2 are equal, but they denote different methods, the second one is when the names N1 and N2 are different, but they denote the same method. In our work the first type of conflicts are more likely to appear. Renaming is possible only when exists a signature that conforms to all the common features that are the subject of renaming. In fact it is not just simple renaming, it involves conversion, reordering, mapping. To summarize, the renaming solution is very complex and requires to take into account the following aspects: i) changing the names of features; ii) adjusting parameter types, number, and order; iii) adjusting the return types; iv) taking into account the modifiers, according to the semantics of reverse inheritance (e.g. private, package, protected, public); v) taking into account the parameter transmission mechanisms (e.g. transmission by value, by address, by reference, in, out, in-out). The possible solutions for matching two feature's (attribute or method) signatures are: (1) in changing the name of attributes, methods and parameters, renaming solutions may be used; (2) for making method return types and parameter types compatible with the correspondent ones, type changes and conversions, casting operations are needed; (3) to match the order of parameters of the target signature, the reordering of parameters must be considered.

B. Circularity

The circularity problem comes from the architectural point of view of an application. In languages like C++ [18, 13], Eiffel [9], Java [1] no circular inheritance class constructs are allowed. So with reverse inheritance, the foster class is not allowed to have a superclass that is already it's direct or indirect subclass. Practically it is impossible to determine at which level will be located the inherited features, because of the infinite circular cycle between classes. Circular inheritance direct, reverse, or any combination of the two is forbidden from this point of view. Example:

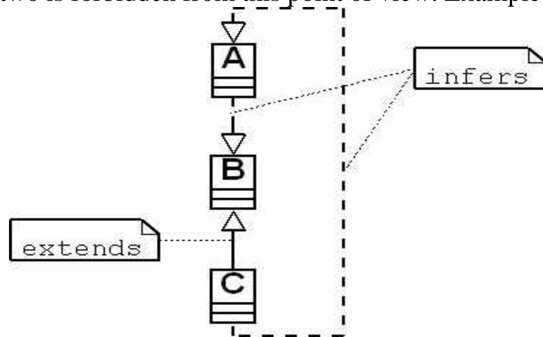

Fig. 3. Circularity

```
class A infers B {}
class B {}
foster class C extends B infers A {}
```

C. Inherited and Factored Features in Foster Class

Another restriction imposed is not to have in the extra record features (explicitly declared or inherited) having the same names as factored features. In the following example such a prohibited situation is presented:

```
class A {
  /* feature that will be inherited
  in foster class */
  int x; }
foster class B extends A infers C {
  /* int x; feature inherited from
class A */
  /* factored feature */
  factored int x;}
class C {
  int x;}
```

The situation above causes that features from class A to be inherited in class B using the E extends A class relationship and features from class C to be factored in the same class B, based on B infers C relationship. If it happens that features with same names to be inherited and factored then a conflict is produced. The solutions that may be adopted in such situations are: the definition of masking rules or the prohibition of such cases.

D. Cascaded Reverse Inheritance

This subsection explains the problems of cascaded reverse inheritance. Cascaded reverse inheritance situation appears when in a class hierarchy there are at least two foster classes on the same path of the inheritance tree and where inheritance and reverse inheritance alternate on the same path from one ancestor class to one descendent class, like in the example code:

```
foster class A infers B {
  /* extra feature */
  int x;}
class B {}
class C extends B infers D {}
class D {}
```

In most object oriented programming languages (Java [1], Eiffel [9], C++ [18,13]), when using direct inheritance, subclasses are defined after superclasses have been defined. So in this context, an inherited feature passes from one class to another through the inheritance relationship. Symmetrically, reverse inheritance should do the reverse thing, because superclasses are created after the subclasses have been created (they belong to libraries that we want to adapt). The order of applying reverse inheritance to the classes is very important, because using different orders it will result different class hierarchies. In the example above lets take a look at the inheritance of x in two situations: situation (1) when the relationship is applied for A infers B declaration first and then for B infers D second, situation (2) when the order of applying is reversed: (1) Feature x from class A is inherited into class B through the reverse inheritance of classes A and B, then from class B to class C by the direct inheritance of classes B and C, and finally from class C to class D. (2) The relationship is applied first to no features of class C (because class C is empty, having

no features) to move into class D. Then the technique is applied to A infers B class relationship and feature x will be inherited in classes B and C because of reverse, respectively direct inheritance. The conclusion drawn from this examples is that in (1) class D inherits feature x and in (2) does not. From this point of view a top-down technique of applying reverse inheritance is preferred in order to keep the normal evolution direction of class hierarchies, but this kind of reverse inheritance usage may be very confusing for the maintainer, because of the described effects.

E. Inheriting Factored Features by Direct Inheritance

Another type of combinations of inheritance and of reverse inheritance are discussed in this subsection. We determine here the influence of reverse inheritance over direct inheritance. We imagine a situation of having a foster class that is also ancestor of another class by to which is linked by direct inheritance:

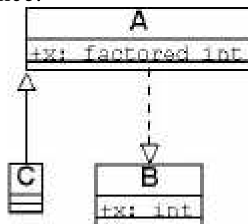

Fig. 4. Inheriting Factored Features

```
foster class A infers B {
  /* factored feature */
  factored int x;}
class B {
  int x;}
class C extends A {}
```

Class A is linked to class B, factoring field x, on the other hand class C is linked to class A, inheriting it's features. The decision of inheriting the factored feature x must be taken. One reason for doing so, is the uniformity of the resulted classes, another reason is that we want to keep factoring features and direct inheritance independent. Direct inheritance works the same with factored or non-factored features.

F. Direct Inherited Features Are Factored

In this subsection a case of fork-join inheritance [4] is analyzed. It is the case of having a direct inherited feature that is subject to factoring.

```
foster class A infers C, D {
  factored int x;}
class B {
  int x;}
class C extends B {
  /* int x - inherited feature
  from class B */}
class D {
  int x;}
```

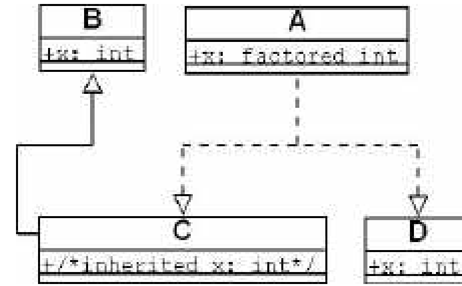

Fig. 5. Direct Inherited Features Are Factored

In the example above foster class A factors feature x from C and D classes. The problem is that in class C, feature x is not declared explicitly but inherited from class B. Certainly accepting this kind of situations induces a lack of clarity in the code but it can be eliminated with the help of an assistance supporting tool. On the other hand class C will have two superclasses, simulating a multiple inheritance construct.

G. Twice Factored Features

Reverse inheritance permits constructs where a class can be forced to factor it's features twice:

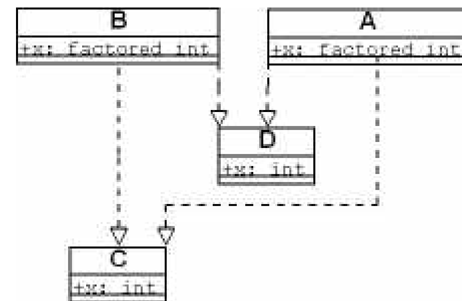

Fig. 6. Twice Factored Features

```
foster class A infers C, D {
  factored int x;}
foster class B infers C, D {
  factored int x;}
class C {
  int x;}
class D {
  int x;}
```

In the sample above, we have two foster classes A and B that infer both C and D classes. So feature x from class C is factored twice it has to be moved into both classes A and B. This class construct is very similar to multiple inheritance constructs, combining two reverse inheritance class relationships multiple inheritance effect is obtained.

IV. RELATED WORKS

In this research area there are works similar to our's, that will be mentioned in this section. In [10] are presented refactoring steps to follow to complete the operation of

finding abstract classes. This work is different from our's: they apply manually refactorings to a system without changing it's behavior, and having no guarantee about the design improve and code integrity; we adapt components changing their behavior and reusing their code, taking into account the restrictions imposed by the context and pointed out in section III.

In [7] a reverse inheritance class relationship for Eiffel language is proposed, discussing ways of factoring out the commonality retroactively from classes that are already developed and there are also analyzed the problems that have to be solved at compile time. This work is very similar to our's but they do not address adaptation or evolution and it is specific to Eiffel.

In [12] the idea of inverse inheritance support in object-oriented programming is developed. There are addressed issued about renaming and combinations of conventional and inverse inheritance are discussed. The combination of the two class relationships allows the features to travel from one class to another through an intermediary class. Among the problems discussed worth mentioning "the fragile superclass problem".

The [4] paper presents hierarchy transformation strategies to transform multiple inheritance hierarchies into single inheritance equivalent hierarchies: emancipation, composition, expansion and variant type. Ideas from this work may be used in making the automatic transformations from state B to state C described in section I.

V. CONCLUSIONS AND FUTURE WORK

In this section conclusions are stated relative to factoring supporting mechanism of reverse inheritance. The case-study that has been presented shows the kind of modifications that may be described using the proposed class relationship. We may conclude from section II that factoring of features is different depending on what type of features we are working with. The two situations have to be treated separately: i) if members are factored then they must be extracted from their original declaration space and put into the foster class; ii) if methods are factored, an abstract method must be created in the foster class. An important issue is to better describe the semantics of reverse inheritance to solve all the arisen problems presented in section III and to propose a homogenous solution like transformations schemes as explained in section I: i) to solve the problem of name conflicts (renaming is a very complex potential solution and it is not the target of this paper); ii) to model a business model related to this purpose; iii) to build the generator that will generate the "library evolution" or the "library adaptation" in order for the former to make reengineering of library and for the letter to reuse exiting code. iv) to use the benefit of the approach of SmartTools or of SmartModels. Another important issue is also to choose the language which will be the target for a first prototype. One of the question related to this aspect is whether we should address languages which support single or multiple inheritance. Typically we may choose between Eiffel or Java. The prototype may be based on existing parsers dedicated to the target language or may use more general tools such as SmartTools [2] or SmartModels. This prototype will work on an abstract syntax tree, which describes the target language extended with reverse inheritance. It's interface could rely on a wizard and be plugged in programming environment such as Eclipse or EiffelStudio.

REFERENCES

[1] K. Arnold and J. Gosling. The Java Programming Language. Sun Microsystems, 3rd edition, USA, 2000.

[2] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joel Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. Smarttools: a development environment generator based on XML technologies. In XML Technologies and Software Engineering ICSE'2001, ICSE workshop proceedings, Toronto, Canada, 2001.

[3] Ciprian-Bogdan Chirila, Pierre Crescenzo, and Philippe Lahire. Towards reengineering: An approach based on reverse inheritance. Application to Java. Research report, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, July 2003.

[4] Yania Crespo, Jos ManuelMarques, and Juan Jos Rodryguez. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In In European Conference on Object-Oriented Programming, 2002.

[5] Peter H. Frohlich. Inheritance decomposed. In Proceedings of the Inheritance Workshop at ECOOP 2002, Malaga, Spain, June 2002.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, 1997.

[7] Ted Lawson, Christine Hollinshead, and Munib Qutaishat. The potential for reverse type inheritance in Eiffel. In Technology of Object-Oriented Languages and Systems (TOOLS'94), 1994.

[8] Bertrand Meyer. Object-Oriented Software Construction 2nd ed. Prentice Hall, 1997.

[9] Bertrand Meyer. Eiffel: The language. http://www.inf.ethz.ch/meyer, September 2002.

[10] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring, 1993.

[11] C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In Conference proceedings on Object-oriented programming systems, languages and applications, pages 407–417. ACM Press, 1989.

[12] Markku Sakkinen. Exheritance - class generalization revived. In Proceedings of the Inheritance Workshop at ECOOP 2002, Malaga, Spain, June 2002.

[13] Herbert Schildt. C++, Manual complet. Editura TEORA, 1997.

[14] Anthony J.H. Simons. The theory of classification, part 7: A class is a type family. Journal of Object Technology, 2(3):13–22, May-June 2003.

[15] Anthony J.H. Simons. The theory of classification, part 8: Classification and inheritance. Journal of Object Technology, 2(4):55–64, July-August 2003.

[16] Anthony J.H. Simons. The theory of classification, part 9: Inheritance and self-reference. Journal of Object Technology, 2(6):25–34, November-December 2003.

[17] Anthony J.H. Simons. The theory of classification, part 10: Method combination and super-reference. Journal of Object Technology, 3(1):43–53, January-February 2004.

[18] Bjarne Stroustrup. The C++ Programming Language Third Edition. Addison-Wesley, 1997.

[19] Antero Taivalsaari. On the notion of inheritance. In ACM Computing Surveys, No. 3, volume 28, September 1996.

[20] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. ACM Transactions on Software Engineering and Methodology, 4(2):146–170, 1995.