A Survey on Reverse Inheritance Class Relationship

Ciprian-Bogdan Chirilă, Pierre Crescenzo, Philippe Lahire, Dan Alexandru Pescaru, Emanuel Ţundrea

▶ **To cite this version:**

Ciprian-Bogdan Chirilă, Pierre Crescenzo, Philippe Lahire, Dan Alexandru Pescaru, Emanuel Ţundrea. A Survey on Reverse Inheritance Class Relationship. Scientific Bulletin of the "Politehnica" University of Timisoara, Transaction on Automatic Control and Computer Science, 2005, 50 (64), pp.45-49. hal-01141800

HAL Id: hal-01141800
https://hal.science/hal-01141800

Submitted on 3 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

BULETINUL STIINTIFIC al Universitatii "Politehnica" din Timisoara, ROMANIA,
Seria AUTOMATICA si CALCULATOARE
SCIENTIFIC BULLETIN of "Politehnica" University of Timisoara, ROMANIA,
Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE, Vol. 50 (64), 2005, ISSN 1224-600X

A Survey on Reverse Inheritance Class Relationship

Ciprian-Bogdan Chirila[*] Pierre Crescenzo[**] Philippe Lahire[**]
Dan Pescaru[*] and Emanuel Țundrea[*]

[*] Department of Computer Science and Software,
University of Timisoara, Faculty of Automatics and, V. Parvan 2, Timisoara Romania
Phone: (+40) 256-404061, Fax: (+40) 256-403214, E-mail: chirila@cs.utt.ro, WWW: http://www.cs.utt.ro/~chirila

[**] 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 *– The reverse inheritance class relationship viewed as the symmetrical of the inheritance class relationship has great potential in class hierarchy reorganization. Classes from different hierarchies can be reorganized getting a new common superclass, factoring common features, thus avoiding data and code duplication.*

Keywords: *inheritance, specialization, reverse inheritance, generalization, exheritance, upward inheritance.*

I. INTRODUCTION

This paper does not intend to bring any contribution but to analyze several reverse inheritance approaches. The development of object-oriented technology, widespread in the software industry, gave birth to a lot of class hierarchy based libraries. The concept of reverse inheritance can help in reusing those classes. In this survey we present the main ideas behind this concept and we analyze the existing solutions to the encountered conflicts and problems. This survey is organized as follows: in the second section are discussed generalities about the concept, the third section is dedicated to definition, sections IV and V deal with interface and implementation exheritance, section VI presents other related works and finally in section VII conclusions are drawn and future works are set.

II. THE CONCEPT

A. History

Reverse inheritance is a class relationship, which seems to have been appeared in the world of object-oriented databases [8]. Pedersen [6] analyzed the concept in the context of object-oriented programming. Later on in [2] ideas about integrating reverse inheritance concept in Eiffel are discussed. In 2002 the concept is revisited and some flaws and also possible solutions are presented [7]. Even if there are a few works dealing with this concept, its semantics was never fully defined nor implemented in a programming language [Ped89,Sak02].

B. Alternative Names

Initially the concept of reverse inheritance was named as upward inheritance in the works of [8] and it was used in homogenizing database schema. It is also known as generalization in the work of [6] where an experimental language was built to define generalization and specialization as symmetrical concepts. The name of reverse inheritance appeared in the paper of [2] where it was proposed as the notion of reverse type inheritance. A very important notion can be considered the superclass of the reverse inheritance class relationship. The superclass practically contains all information related to reverse inheritance semantics in a class hierarchy. It is known also as generalizing class [6], foster class [2] or exheriting class [7].

C. Principle of Reverse Inheritance

With ordinary inheritance, the superclass exists first and then the subclasses are created by refinement. Conversely, with reverse inheritance the process is backward: starting from several subclasses there can be designed a common superclass. In any class hierarchy there should be no difference whether it was created by ordinary or reverse inheritance. It is argued in [6] that it is more natural to design concrete specialized classes and after that to notice commonalities and to create a more abstract class. So ordinary inheritance implies a top-down design, while reverse inheritance a bottom-up one.

III. DEFINITION

D. The Intension and Extension of a Class

In [6] is presented a simplification of the object concept. The intension of a class is the set of properties through which it is defined. An example is given in this sense: the

"mammal" concept is analyzed. The intension of this concept refers to real-world properties like: these animals have mammae which secrets milk as nourishment for their young. By extension of a class we mean all the phenomena that include those properties. Back to the analyzed example it can be considered that the neighbor's dog belongs to the extension of the mammal concept.

E. Specialization and Generalization

Specialization can be defined in terms of intension and extension of a concept. A concept $C_{special}$ is a specialization of a concept C, if all phenomena of $C_{special}^{extension}$ belong to $C^{extension}$ [6]. Concept worker is a single specialization of concept employee, since all workers have all properties of employees and eventually some extra. A worker can take the place of an employee but not necessarily the other way around. Formally this can be expressed like: a concept $C_{special}$ is a single specialization of a concept C iff $x \in C_{special}^{extension} \Rightarrow x \in C^{extension}$. There can de defined also the notion of multiple specialization in the same way: a concept is a multiple specialization of a set of other concepts if it is a single specialization of each concept in the set [6]. Concept calculator-watch is a specialization of both concepts calculator and watch. Calculator-watch fulfils the properties of calculator and watch. Formally, a concept $C_{special}$ is a multiple specialization of $C_1,...,C_n$ iff $x \in C_{special}^{extension} \Rightarrow \forall i \in 1...n : x \in C_i^{extension}$ [6].

Generalization can be defined also in terms of intension and extension of a concept [6]: a concept $C_{general}$ is a single generalization of a concept C if all members of $C^{extension}$ are members also in $C_{general}^{extension}$. This means that all phenomena belonging to $C^{extension}$ will belong also to $C_{general}^{extension}$. Concept employee is a generalization of concept worker since every worker is an employee. Formally $C_{general}$ is a generalization of concept C iff $x \in C^{extension} \Rightarrow x \in C_{general}^{extension}$. As in the case of specialization there is multiple generalization. A concept is a multiple generalization of a set of other concepts if it is a single generalization of every concept in the set. For example the concept of employee is a generalization of worker, manager, security guard, secretary, because all are employees. In formal notation $C_{general}$ is a generalization of $C_1,...,C_n$ iff $\forall i \in 1...n, x \in C_i^{extension} \Rightarrow x \in C_{general}^{extension}$.

F. Cardinality

Ordinary inheritance can be single or multiple, so it is the case for reverse inheritance, as it was defined in the previous section. Single reverse inheritance supposes that there is a single subclass and a foster class. In figure 1 we have the example of a Dequeue [6], which shows that single reverse inheritance can be useful.

Fig. 1. Dequeue Sample

The DEQUEUE class models a double ended queue having the classical stack operations at both ends (push, pop, top, push2, pop2, top2, empty).

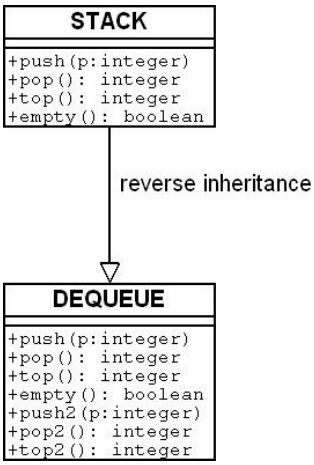

Fig. 2. Dequeue Class Diagram

Later a new class is needed in order to model the behavior of a simple stack. So class STACK is designed exheriting only the operations push, pop, top, empty. In this sample single reverse inheritance was used to create a new type from an existing one.

IV. INTERFACE EXHERITANCE

G. Common Features

By common features we mean those features from the subclasses, which have the same semantics and are subject of factorization in the foster class. Thus code and data can be reused without duplication. On the other hand the possibility of common features specialization is available by subclassing. Still there are problems related to name, type, signature, assertion conflicts. These common features in a normal inheritance top-down equivalent design would be the ones inherited in each subclass.

H. Concrete vs. Abstract Generalizing Classes

In [6] it is emphasized that interface exheritance is the most simple. As mentioned in [7], the integration of interface exheritance in Java can be done with minimum of effort because of the notion of "interface" they introduced in the

language. A Java interface consists in a set of abstract methods [1]. It can be considered as a pure abstract class. An abstract class in Java may contain abstract methods having no implementation, just signature and also concrete methods with implementation. We note also that interfaces can be created by specialization of several multiple interfaces, they can be implemented by several subclasses and their methods are all public. It is suggested that interfaces could be defined by generalization of classes and other interfaces. Not all languages possess such an interface concept like Java does, so we have to use the class concept as generalization classifier. For those languages is proposed [7] the idea of generalization into fully abstract classes (e.g. Eiffel [4], C++ [9], Java [1]).

I. Type Conformance Between Subclass / Superclass

Related to interface exheritance issue, in [6] it is demonstrated using an experimental language that from the point of view of type conformance, there are no conflicts introduced in a class hierarchy having subclasses / superclasses introduced by inheritance / reverse inheritance. The main idea of the demonstration is to prove using formalisms that the feature set of the generalizing class contains at most the intersection of the feature subclasses sets. Before proving, some notations are necessary:

$$A^{methods} = \{m_1,...,m_n\}$$

denotes the set of methods of class A. Class A is defined as generalization of classes $B_1, B_2,..., B_k$ removing methods $m_1, m_2,..., m_n$. To prove that B_i ($i \in 1..k$) conforms to A, means that class A method set is a subset of those of any instance of class B_i($i \in 1..k$). We use the following formalism:

$$A^{methods} = \bigcap_{i=1}^{k} B_i^{methods} \setminus \{m_1,...,m_n\}$$

So it is demonstrated that A is a superclass of B_i ($i \in 1...k$), so the conformance rule is valid. In the [2] definition of semantics a type conformance rule is set. The type of subclasses has to conform to the type of superclass. From their point of view the superclass type is a generalization of the subclasses types. It can imply type intersection or type union, depending on the type definition. In [6] is discussed about the intension and the extension of an object. Referring to these two conceptual aspects of an object they consider that if a type is a set of features than the type of the superclass should be their intersection. If the type is considered as a set of objects, then the superclass type of the generalizing class will be a least the union of the subclass types.

V. IMPLEMENTATION EXHERITANCE

It is mentioned in [7] that implementation exheritance refers mostly to attributes and methods. In the case of attributes some type and visibility conflicts are foreseen.

J. With No Virtual Methods

Regarding implementation exheritance in [6] first it is considered the case of languages which have non-virtual methods. The idea proposed is to exherit in the foster class the implementation of one of the subclasses (it is named in [7] as principal subclass). Of course, the compiler will have to take care of methods to be possible to execute in the context of foster instances. If there is only one subclass the choice is implicit. If there are multiple subclasses it is proposed that the programmer should select the one from which the foster class gets it's implementation. It can not be made automatically because the programmer knows better the implementations from the subclasses and ha can make an optimal decision. Also it is taken into account the fact that the implementation selected for the foster class can be used in creating new specializations. Also a new implementation can be provided for the foster class and has the advantage of avoiding dependencies on implementations of other classes.

This approach is severely criticized in [7] since the exherited implementations from the principal subclass will be inherited in all the subclasses, thus changing their original behavior. It is admitted that no other exherited classes, except the principal subclass would be the subclass of the foster class. In the case of adding new implementation in the foster class it is mentioned that no exherited class will be the subclass of the foster, except some cases based on coincidence.

K. With Virtual Methods

In [6] the case of object-oriented programming languages which support virtual methods the things are not so problematic. Virtual methods are refined differently in the subclasses and it is intended that the implementation of the foster class to contain the common behavior among them. There are three cases analyzed:

i) When there is no common behavior, there should be only empty methods. All the eventual future subclasses of the foster class will have to implement these methods.

ii) If all implementations exhibit the same behavior the implementation for the foster class can be taken from the principle subclass used in the non-virtual case.

iii) When there is some common behavior, the programmer has to choose one implementation and it has to be the one which contains the common behavior.

In [7] it is proposed that either exheritance should be restricted to interfaces only or to use a feasible solution for implementation exheritance. The solution proposed is to let the programmer select for each exherited method the suitable implementation from the different subclasses. With this solution we have the problem of references, meaning that such an exherited method needs it's attributes and methods that depends on.

L. Implementation Problems

In [7] it is noted that exherited methods may contain type verifications which in the normal context of the subclasses works normally, but in the context of the foster class it may fail. A special problem pointed out is the one generated by the invariants. They can be checked only at runtime and

they must be stronger in the foster class than its correspondents in the subclasses. Preconditions and postconditions

M. Name Conflicts

In [7] is presented the problem of name conflicts. It is encountered when exheriting features (attributes or methods) with the same semantics having different name or when two different features have the same name. The first is named "lost friends" in [7] and it can be resolved using syntax extension. The second case is named "false friends" [7] and such features have not to be exherited since they are not the same [6]. Both conflicts can not be automatically detected, so the programmer must declare them explicitly. In [8] there is presented a different approach to this problem. It is used a meta-class for the foster class model, which integrates the renaming mechanism.

VI. RELATED WORKS

Among other class reorganization techniques we can mention multiple inheritance, like-type class relationship, traits, mixins.

An alternative to reverse inheritance class relationship would be to use class hierarchy transformations presented in [5] like variant types or simulation with monitor class and flags.

In UML [10] there is only one notation for both generalization and specialization class relationships. There can be used arrows originated from the most specialized class towards the most general one.

In [OJ93] there is presented a systematical method in building abstract superclasses using refactoring techniques. The process involves adding function signatures to the superclass, making the function bodies compatible, moving variables, migrating common code to the superclass. It is admitted that the drawback of such a methodology is that arbitrary recfactorings may affect the original design of the classes even the behavior is unchanged.

VII. CONCLUSIONS AND FUTURE WORK

Inheritance and reverse inheritance are complementary class relationships. They are not redundant because of the two top-down and bottom up design methods and because of the class adaptation mechanisms of reverse inheritance. Some adaptations presented are very particular solutions to a very general problem. So adaptation mechanism could be extended in more general sense. Exheritance implementation seems to need local solutions relative to a concrete programming language. The problem of name conflicts of reverse inheritance is the same as in multiple inheritance. Depending on the programming language in which the concept is integrated into, it seems that there are decent solutions in this sense.

We consider that reverse inheritance has a great potential. It can be seen as a class relationship in the object-oriented languages. Also it could be used as a class hierarchy reorganization tool with restricted adaptation and limited evolution purposes, in which case reverse inheritance is volatile. It's existence is resumed to only one phase of the design. Reverse inheritance represents a great potential composition mechanism for weaving concerns. However the last two ideas were never approached in the literature.

As future work we propose the integration of the reverse inheritance concept in the Eiffel programming language. We consider that this concept can be integrated better in the philosophy of Eiffel. We have the renaming facility already existing in Eiffel, the presence of multiple inheritance and future multiple reverse inheritance keeps the symmetry of the language. Of course there are drawbacks like the problem of assertions in interface exheritance.

ACKNOWLEDGMENTS

This paper is part of the PhD thesis on reverse inheritance class relationship, developed in collaboration with the OCL team from the I3S (Informatique Signaux Systèmes de Sophia) Research Institute affiliated to University Sophia-Antipolis of Nice, France. Also we want to thank professor Markku Sakkinen from the University of Jyväskylä, Finland for the valuable ideas and feedback he gave us.

REFERENCES

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

[2] 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.

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

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

[5] Yania Crespo and Jos Manuel Marques and Juan Jos Rodryguez, On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies, In European Conference on Object-Oriented Programming, Malaga, Spain, 2002.

[6] 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.

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

[8] Michael Schrefl and Erich J. Neuhold. Object class definition by generalization using upward inheritance. In IEEE Transactions, 1988.

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

[10] UML Superstructure Version 2.0, www.omg.org/uml, October, 2004