On transformation strategies from multiple inheritance to single inheritance. A comparative approach[*]

Juan José Rodríguez
juanjo@infor.uva.es
Dpto. de Informática
Universidad de Valladolid
España

Yania Crespo[†]
yania@infor.uva.es
Dpto. de CC. de la Comp.
Universidad de La Habana
Cuba

José Manuel Marqués
jmmc@infor.uva.es
Dpto. de Informática
Universidad de Valladolid
España

Abstract

In this paper we briefly present some solution strategies for situations in which it is necessary to transform multiple inheritance schemes into single inheritance or non-inheritance "equivalent" schemes. The strategies are divided into basic strategies and combined strategies. The mechanisms presented are comparatively analyzed in the light of some ideal characteristics to be accomplished by hierarchy transformation strategies.

Keywords: object oriented, single inheritance, multiple inheritance, conversion strategies

1 Introduction and Motivation

It is possible to describe 3 situations in which it would be useful to have available some transformation mechanisms from multiple inheritance to single inheritance:

- To extend a single inheritance language with multiple inheritance constructions.
- To implement a multiple inheritance based model in a single inheritance language.
- To translate from a multiple inheritance language to a single inheritance language.

The second situation is specially interesting for Software Engineering. This is because of its usefulness in CASE tools, in automatic prototyping environments and also in those environments for aiding in reuse from different abstraction levels. The automatic realization of this mechanism is useful from the previous three points of view because:

- It can be incorporated into Object Oriented Design (OOD) CASE tools in order to support code generation independently of the target language.
- It can be used in automatic prototyping environments if they encourage multiple inheritance as a specification resource, or some other specification resource leading to a multiple inheritance design solution, if it one wishes to obtain a prototype coded in a single inheritance language[1].
- It can be incorporated into environments supporting reuse from different abstraction levels. In this case it will be possible to reuse a multiple inheritance based OOD in some other project requiring a particular target language that does not allow multiple inheritance. Functionality of *assets* repositories and libraries will also be improved[2].

Object Oriented Programming Languages and Inheritance

Figure 1 represents some examples of language categorized according to the presence of inheritance relations as language construction.

[*]This work has been partially supported thanks to CICYT project TIC97-0593-C05-05 as part of the MENHIR project.

[†]Yania Crespo is a research student thanks to the Instituto de Cooperación Iberoamericana (ICI).

[1]An example for this situation is found in the code generation tool for OO-METHOD conceptual models [13] (Universidad Politécnica de Valencia, MENHIR project).

[2]An example for this situation is the repository based in complex reuse structures currently under development in the MECANO project [7] (Universidad de Valladolid) as a part of the MENHIR project.

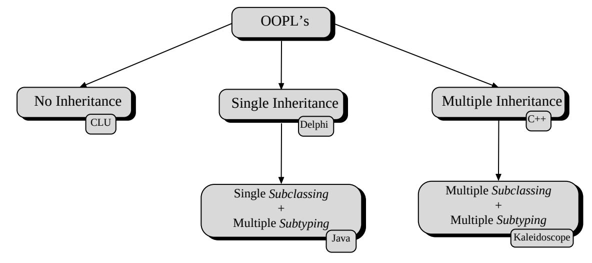
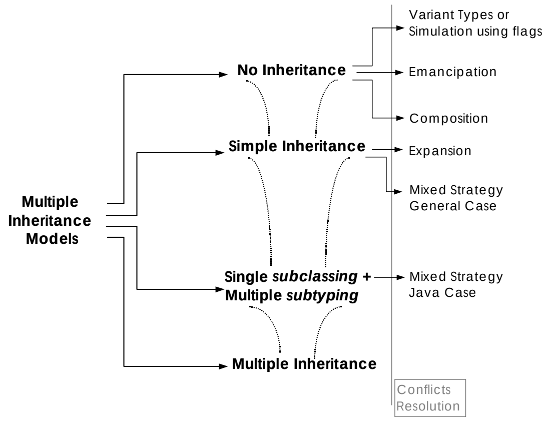


Figure 1: Some languages on the presence of inheritance.



Multiple Inheritance Models
No Inheritance
Simple Inheritance
Single subclassing + Multiple subtyping
Multiple Inheritance
Variant Types or Simulation using flags
Emancipation
Composition
Expansion
Mixed Strategy General Case
Mixed Strategy Java Case
Conflicts Resolution

Figure 2: Languages and transformation mechanisms.

CLU [8] is class based language, objects+classes without any inheritance mechanism (following the Wegner classification [19]). Delphi [5] is an application development tool based in Object Pascal, an object oriented language (objects+classes+inheritance), from which it acquired the characteristic of just allowing single inheritance[3] (Modula-3 [3] is another example of this case). Eiffel [11], C++ [18] y CLOS [16] are examples of languages supporting multiple inheritance of classes.

Java [1] is a language that separates type and class notions into 2 structures: interfaces and classes as implementations of interfaces. For Java interfaces, multiple derivations are allowed, but for Java classes, just single derivations are allowed. A class can implement multiple interfaces. On the other hand, Kaleidoscope [6] is an example of a language that separates the same notions but allowing multiple derivations for both of them. In the following section these language categories will be related to the transformation strategies that apply to each case. Figure 2 shows this relation schematically. What the dotted line means is that for those languages it is always possible to apply the preceding methods even through they don't fully exploit all the language characteristics.

2 Conversion strategies from multiple inheritance into single inheritance

We propose that an ideal transformation strategy from multiple inheritance schemes into single inheritance schemes should:

- Maintain as far as possible the originally modeled inheritance hierarchic classification.
- Respect polymorphic assignments and behaviour even if the languages are strongly statically typed.
- Avoid (as far as possible) excessive code repetition and problems of coherence maintenance.

These aspects will be considered as a reference for analyzing each transformation strategy. Their complexity and the kind of language their apply to will be also evaluated.

2.1 Basic strategies

The problem of transforming a multiple inheritance hierarchy into a single inheritance "equivalent" hierarchy can be tackled using the following basic strategies:

- **Emancipation:** all the inheritance relations of a class are eliminated, and all the inherited properties are included as its own resources. This strategy is similar to the application of *flattening* to a class [10, 12]. In this operation it must be taken into account that the coexistence of various versions of the same method (the ancestors versions) could be necessary so that the original calls to super methods can be translated into local calls. This requires the methods to be renamed, the calls to be modified and so on.

[3]This description corresponds with Delphi 3 because currently Delphi 4 is situated in the same case as Java.

- **Composition:** the inheritance relations are transformed into composition relations. According with Meyer [12], when it is the case that a class B needs a facility from some other class A there are 2 possibilities. Is B an heir of A, or a client of A? Even though there is a marked difference between *is-a* and *has-a* relationships, sometimes it is possible to change an *is-a* relationship by a *has-a* relationship (with the loss of benefits this implies). The calls to super methods are solved delegating in the composite object. As a consequence, the methods body must be modified.

- **Expansion:** the multiple inheritance graph (DAG) is expanded into a tree (a forest, generally). Hence, in the new graph there are only single inheritance relations. In this method the transformation is achieved without loss of information thanks to the replication of classes [9, Chapter 4].

- ***Variant* type or its simulation with a monitor class and flags:** starting from root classes on the hierarchy, a complex structure including all the properties of its descendant classes is created for each one. These properties are dispatched according to the current object. The structure is a *variant* type (in the case where the target language has a type construct for *variant*) or a monitor class simulating *variants* with flags. Unlike the emancipation process, a flattened class is not obtained for each original class. The result is a unique complex structure describing the objects of one or other class depending on a condition.

Table 1 shows a comparative evaluation of the methods named basic strategies.

The analysis of basic strategies leads us to the conclusion that none of them is completely satisfactory by itself because there is a great loss of the original model and/or some (or all) of the ideal characteristics already exposed are broken.

2.2 Combining strategies

To the present time there are two proposals for combining strategies and they are classified as: the Java case and the General case [15]. The special distinction for Java is justified because, although it is a particular case, it is interesting and representative and it serves as a basis for considering the general case. The Java case tries to cover those languages with single class inheritance but multiple interface derivation.

For each one of these cases some variants can be described according to the basic strategies that they choose to combine. We shall now briefly describe a variant combining interfaces with emancipation (variant 1) and another that combines interfaces with emancipation and composition (variant 2) for the Java case. There are equivalent variants for the general case but they change interfaces by the use of expansion.

Given an original hierarchy, this can be represented using Java interfaces maintaining all inheritance relationships, but neither non-constant attributes nor methods bodies can not be included in these interfaces. Therefore, it is necessary that some classes implement those interfaces. In order to simulate the existence of attributes using interfaces there must be defined methods signatures for getting and setting the required attributes.

In the variant 1, classes for implementing interfaces are obtained as a result of the emancipation process. The polymorphic assignments are achieved by declaring the variables managing objects with the interfaces. The corresponding polymorphic behavior is achieved by creating these objects with the emancipated classes. The emancipation process generates methods duplication in the set of resultant classes.

In the more complex variant 2, a new group of classes resulting from the application of the composition process to the original hierarchy is introduced. The whole process includes filtering emancipated and aggregated classes in the following way: keep in the classes resulting from composition only the methods and according to this, keep in the emancipated classes only the attributes. The interfaces obtention from the original hierarchy does not change.

Relating the 3 groups we have: each method class implements its corresponding interface

Comparing basic strategies	**Emancipation**	**Composition**	**Expansion**	**Variant**
Hierarchy preservation	None	Some kind of composition hierarchy	Much	None
Code duplication	Much	None	Some	None
Coherence maintenance	No problem	Some problem	Some	No problem
Polymorphic assignments and behavior	Problems. It could be simulated but with problems of overhead and coherence maintenance.	Problems. It could be simulated without overhead but it leads to problems with coherence maintenance and polymorphic behavior.	Problems only in some cases. For this cases it could be simulated but it leads to coherence maintenance problems.	No problems but with nuances. OO language compilers checkers and automatic dispatchers are lost. They must be manually simulated.
Application area	Languages with Objects + Classes	Languages with Objects + Classes	Languages with Objects + Classes + Inheritance	Languages with Objects + Classes
Application complexity	Simple	Simple	Complex	Fairly complex

Table 1: Evaluation of basic transformation strategies

and each attribute class is a component of its corresponding method class (this is the same as to say that each method class has an attribute that is representing an object of the corresponding attribute class). When an object of a method class is created, the attributes of its components classes keep null, i.e. only the object of attributes directly related with it is created. The interfaces are shells for dispatching calls to objects.

Between method classes and interfaces an association named *Delegates* is established which indicates which object is delegating in its corresponding method class. This association is needed in order to maintain the self reference and to work without loss of polymorphic behavior. Calls to methods of the class itself are always sent to the delegator.

This is a brief strategy evaluation. For languages with single subclassing and multiple subtyping, all basic and combined strategies are applicable but the former, described as variants of the Java case, are the ones which better exploit the language characteristics. Both variants 1 and 2 resolve correctly the original inheritance classification (using interfaces) and preserve polymorphic assignments and behavior without coherence maintenance problems. Variant 1 presents code duplication but variant 2 does not.

All of the mentioned strategies are algorithmically detailed and presented with examples in [15].

3 Some related works

The strategy here named expansion was proposed by Marqués in [9]. The strategy named emancipation takes the idea from the notion of the *flat* form of a class [10], in the same way as the composition strategy is based on the works of Stein concerning delegation [17]. The strategy which is called "*variant* types or ..." is an idea starting from part of the automatic codification of conceptual OO-METHOD models works [14]. The work on types of Cardelli and Wegner [2, 4] were also a basis for its conformation. In [15] can be found a detailed and broader description of related works.

4 Conclusions y future work

In this paper several approaches to the problem of transforming multiple inheritance hierarchies into single inheritance hierarchies are analyzed. These approaches are coarsely divided into: basic strategies and combined strategies. Some strategies were briefly presented and each of these were analyzed according to the defined ideal characteristic to be accomplished by a transformation mechanism, to its application complexity and to its application area for different languages.

An immediate work to be developed consists in the analysis of the strategies in the light of real examples. This will allow us to make a comparison of them but according to how they fixed to the type of model that is wanted to transform. As described in [14], when modeling with OO-METHOD, several situations appear that can lead to multiple inheritance solution designs. In the current version of the generator similar situations to the Java case variant 1 and the basic strategies named composition and "*variant* types or ..." are used. Exploiting this coincidence, we think it would be interesting to try the fusion of the tendencies OO-METHOD CASE follows with the study that is presented here in order to obtain a more concrete evaluation of strategies.

Experimental and test works developed or in progress take into account other important aspects like creation methods, access control and abstract classes that are not presented here for reasons of brevity .

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Java Series. Sun Microsystems, 1996.

[2] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, 1989.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), Dec 1985.

[5] Delphi 4. Object pascal language guide. Inprise Corporation, available on the web: http://www.inprise.com/delphi.

[6] B.N. Freeman-Benson and A. Borning. The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society International Conference on Programming Languages*, pages 174–180, April 1992.

[7] F.J. García, J.M. Marqués, and J.M. Maudes. Mecanos as basis of compositional/generative mixed reuse model. *II European Reuse Workshop, Madrid, Spain*, November 1998.

[8] B. Liskov. A history of CLU. *SIGPLAN Notices*, 28(2), March 1993. Contents of History of Programming Languages Conference (HOPL-II), Cambridge, Massachusetts, USA, April 20-23, 1993.

[9] J.M. Marqués. *Jerarquías de herencia en el diseño de software orientado al objeto*. PhD thesis, Universidad de Valladolid, 1995.

[10] B. Meyer. Tools for a new culture: Lessons from the design of the Eiffel libraries. *CACM*, 33(9), Sept 1990.

[11] B. Meyer. *Eiffel: the language*. Prentice-Hall Object-Oriented Series, 1991. second revised printing 1992.

[12] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[13] O. Pastor, E. Insfrán, V. Pelechano, J. Romero, and J. Merseguer. OO-METHOD: An OO software production environment combining conventional and formal methods. In *Conference on Advanced Information Systems Engineering (CAiSE'97). Barcelona, Spain*, 1997.

[14] V. Pelechano. Fundamentos metodológicos para el tratamiento de la herencia múltiple en un entorno de producción automática de software. Aspectos de notación, semántica y generación automática de código. Technical report, Departamento de Sistemas Informáticos y Computación (DSIC). Universidad Politécnica de Valencia, 1998.

[15] J.J. Rodríguez, Y. Crespo, and J.M. Marqués. Transformación de jerarquías de herencia múltiple en jerarquías de herencia sencilla. Technical Report TR-GIRO-03-98, Departamento de Informática, Universidad de Valladolid, 1998.

[16] G.L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[17] L. Stein. Delegation is inheritance. In *Proceedings of OOPSLA'87*, 1987.

[18] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[19] P. Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA'87*, 1987.