

# On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies

Yania Crespo<sup>1</sup>, José Manuel Marqués<sup>1</sup>, and Juan José Rodríguez<sup>2</sup>

<sup>1</sup> Dept. of Computer Science, Universidad de Valladolid, Spain  
{yania, jmmc}@infor.uva.es

<sup>2</sup> Dept. of Civil Engineering, Universidad de Burgos, Spain  
jjrodriguez@ubu.es

**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.

## 1 Introduction and Motivation

It is possible to describe three 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 transformation 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 (such as dynamic and role classes [17]), if it one wishes to obtain a prototype coded in a single inheritance language.
- 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 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 they apply to will be also evaluated.

Figure 1 shows schematically the relation between languages and transformation strategies. What the dotted arcs mean 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.

The same Figure 1 shows a proposal for dealing with conflict resolution in the moment of the transformation. That's what the dotted line means in the figure. In [9] was stated that, rather than trying to find *the* universal method

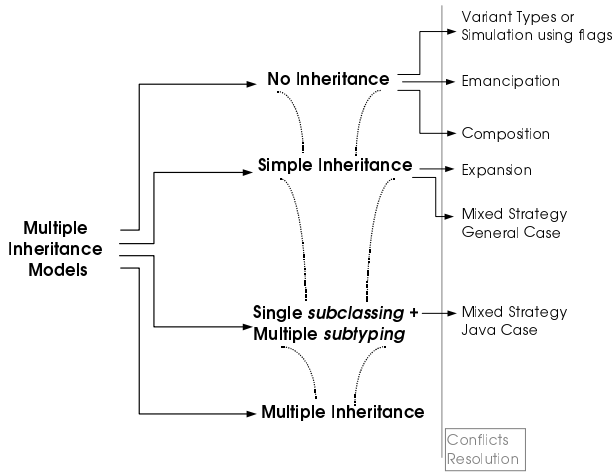


Fig. 1. Languages and transformation mechanisms.

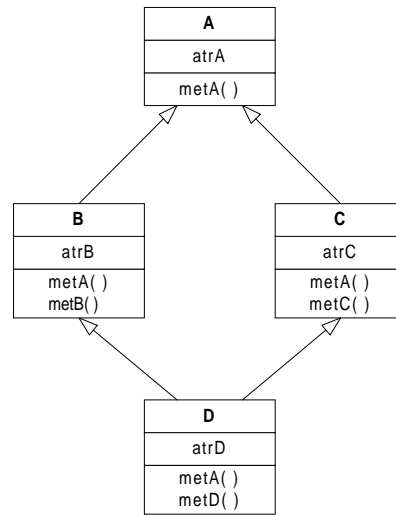


Fig. 2. A multiple inheritance hierarchy. Running example.

for multiple inheritance conflict resolution, it should be considered to apply different resolution methods to different kinds of problems. Our proposal includes this idea. The selected method for conflict resolution determines some minor transformations in the translation mechanism. This is a versatile way for conflict resolution. For instance, in the presence of repeated inheritance, duplication can be obtained by generating two (or more) renamed attributes, unification can be obtained by generating just one attribute and even more complex decisions can be made, combining duplicated attributes by the generation of two (or more) renamed attributes and a function with the original attribute name.

In [3] conflict resolution methods are classified in two main classes: guided resolution (in C++, Eiffel) and automatic resolution (in CLOS). The way of dealing with conflict resolution proposed in Figure 1 is in the mid way between this two classes. It is guided resolution but previous to transformation time (not necessarily in specification or programming time) and it is automatic resolution at transformation time.

## 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, composition, expansion and *Variant* type or its simulation with a monitor class and flags.

The transformations will be illustrated through the classical example of a diamond hierarchy, Figure 2. There are four classes in the example. Each class introduces a new attribute and a new method. All classes define their own version of the method `metA`.

- **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 [7, 8]. 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. Figure 3 shows the example hierarchy transformed by emancipation.
- **Composition:** the inheritance relations are transformed into composition relations. According with Meyer [8], 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 into a *has-a* relationship (with the loss of benefits this implies). The calls to super methods are replaced by delegation. As a consequence, the method bodies must be modified. Figure 4 shows the example hierarchy transformed by composition.
- **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 [6, Chapter 4]. Figure 5 shows the example hierarchy transformed by expansion.

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

- **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. Figure 6 shows the example hierarchy transformed by *variant* . . . .

Table 1 shows a comparative evaluation of the methods named basic strategies. The four basic strategies are detailed presented in algebraic notation and with UML graphical examples in [12].

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 listed at the start of Section 2 are broken.

## 2.2 Combining strategies

In this section we present two proposals for combining strategies. They are going to be referred as: the Java case and the General case. 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 (such as: Java itself, Delphi, Modula-3).

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.

- **Java case:** Given an original hierarchy, it can be represented using Java interfaces maintaining all inheritance relationships, but neither non-constant attributes nor method bodies can 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.  
**Variant 1:** 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

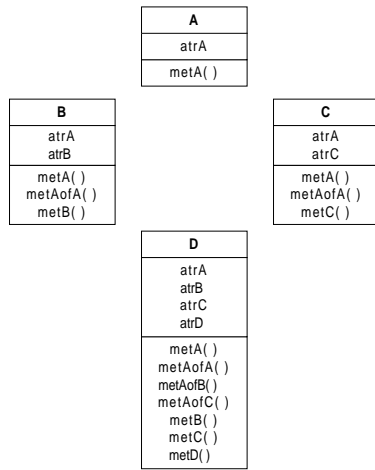


Fig. 3. The hierarchy after the emancipation transformation.

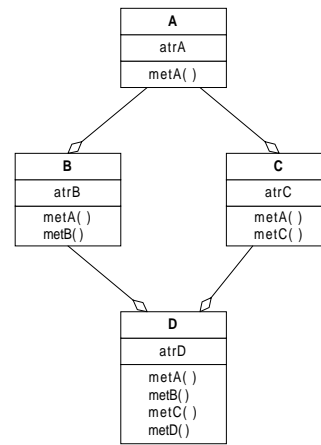


Fig. 4. The hierarchy after the composition transformation.

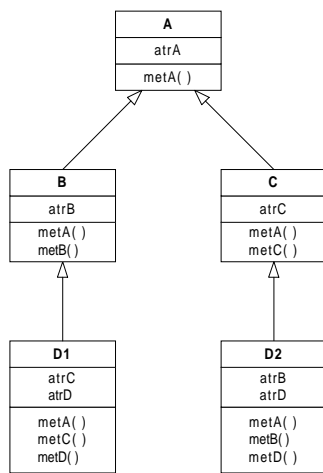


Fig. 5. The hierarchy after the expansion transformation.

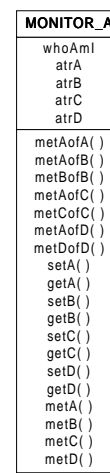


Fig. 6. The hierarchy after the *variant* . . . transformation.

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. Figure 7 shows the example hierarchy transformed by this variant.

**VARIANT 2:** 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. Figure 11 shows the example hierarchy transformed by this variant while Figure 8 shows object diagrams for classes B and D.

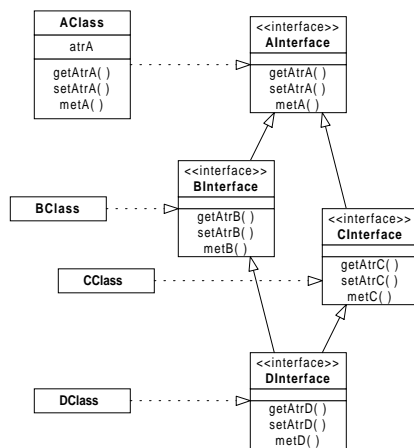
Relating the 3 groups we have: each method class implements its corresponding interface and each attribute class is a component of its corresponding method class (this is the same as to say that each method class has an attribute that is representing an object of the corresponding attribute class). When 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.

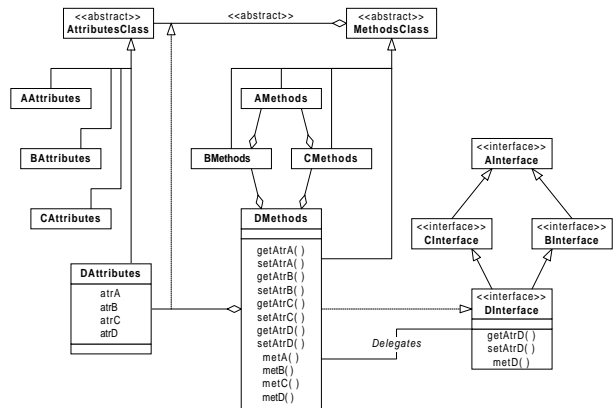
- **General case:** it is similar to the Java case but replacing the interface hierarchy with a hierarchy obtained from the expansion of the original hierarchy.

Comparing combined strategies	Java case (variant 1)	Java case (variant 2)	General case (variant 1)	General case (variant 2)
Hierarchy preservation	as interfaces	as interfaces	Much (as in expansion)	Much (as in expansion)
Code duplication	Some	None	Some	Some (but less than in variant 1)
Coherence maintenance	No problem	No problem	No problem	No problem
Polymorphic assignments and behavior	No Problems.	No Problems.	No Problems.	No problems.
Application area	Languages with Objects + Classes + Simple Subclassing + Multiple Subtyping	Languages with Objects + Classes + Simple Subclassing + Multiple Subtyping	Languages with Objects + Classes + Inheritance	Languages with Objects + Classes
Application complexity	Simple	Simple	Complex	Fairly complex

**Table 2.** Evaluation of combined transformation strategies



**Fig. 7.** The hierarchy after the Java case 1 transformation.



**Fig. 8.** The hierarchy after the Java case 2 transformation.

**Variant 1:** it is similar to the Java case but the relation between the expanded hierarchy and the emancipated classes must be a composition relation and the expanded classes methods must delegate in the emancipated classes. The problems of polymorphic assignments and behaviour and coherence maintenance are solved by creating objects of all expanded classes. Given an original class, the objects created from expanded classes corresponding to the original class will share a common object (created from the corresponding emancipated class). Figure 9 shows the example hierarchy transformed by this variant.

**Variant 2:** it is again similar to the Java case with the ideas commented in Variant 1 of this General case. The expanded classes have a composition relation to the corresponding class resulting from the composition process, and these ones with the appropriated class of the emancipated classes. Among all the classes in the expanded hierarchy that correspond to an original class with multiple inheritance, one is selected to act as the delegator. Figure 10 shows the example hierarchy transformed by this variant while Figure 12 shows the object diagram for class D.

Table 2 shows a comparative evaluation of the methods named combined strategies in their variants.

For languages with single subclassing and multiple subtyping, all basic and combined strategies are applicable but the 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 some code duplication but variant 2

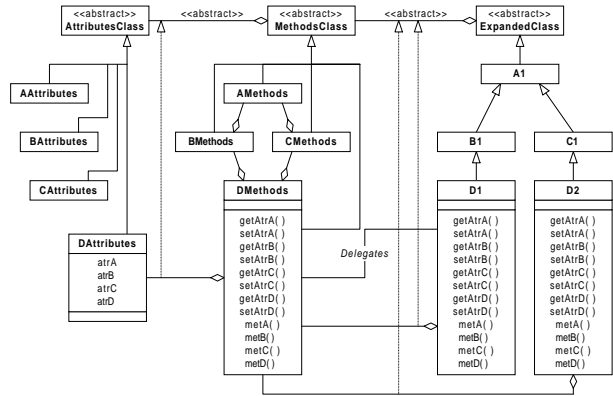
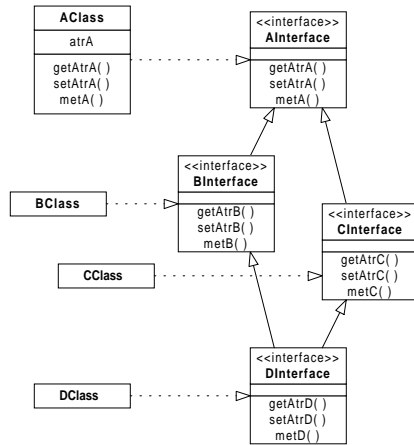


Fig. 9. The hierarchy after the General case 1 transformation. Fig. 10. The hierarchy after the General case 2 transformation.

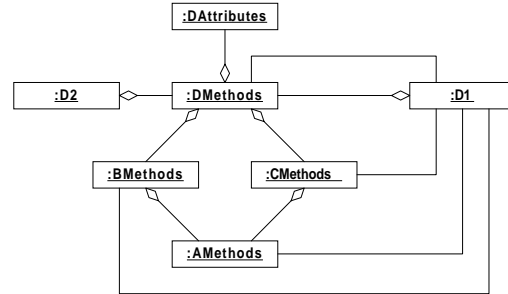
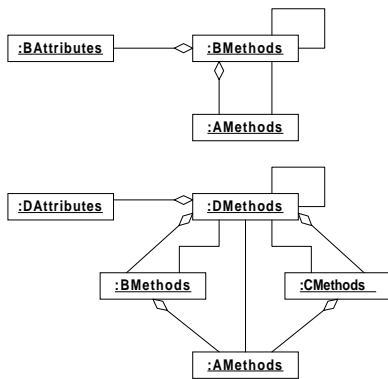


Fig. 11. Objects corresponding to classes B and D in Java case 2 transformation. Fig. 12. Objects corresponding to class D in General case 2 transformation.

does not. In the General case we found always some code duplication because of the hierarchy expansion. Anyway the duplicated code is always a delegation code (a wrapper calling to other object).

All of the mentioned variants of combined strategies are algorithmically detailed and presented with examples in [12].

### 3 Some related works

The strategy here named expansion was described algorithmically in algebraic notation in [6]. The strategy named emancipation takes the idea from the notion of the *flat* form of a class [7], in the same way as the composition strategy is based on the works of Stein concerning delegation [13]. The strategy which is called “variant types or ...” is an idea starting from part of the automatic codification of conceptual OO-METHOD models works [10, 11]. The work on types of Cardelli and Wegner [1, 2] were also a basis for its conformation.

Jamie [15] is a pre-processor for Java that provides automating delegation as an alternative to multiple inheritance. In their approach, the programmer must decide to which object delegate. Its purpose isn’t the automatic translation, but could be helpful in future versions of the implementation of our solutions.

J2C++<sup>3</sup> allows access C++ objects from Java. It associates to every C++ class a Java proxy class. J2C++ only supports single inheritance. Our solution could be used to solve this handicap.

Bruce<sup>4</sup> is an Eiffel to Java translator. At the moment, little information is available, but they “map each (non-generic) Eiffel class to a corresponding Java interface” and “choose to select a maximal subtree of the Eiffel hierarchy of effective classes for direct translation to a Java class hierarchy, the remaining Eiffel effective classes need to be flattened”. Nevertheless, our solutions can avoid method duplication and aren’t constrained to Java.

<sup>3</sup> <http://www.alphaworks.ibm.com/formula.nsf/toolpreview>

<sup>4</sup> <http://www.mri.mq.edu.au/dotg/projects/bruce>

The paper [14] present a work on simulating multiple inheritance in Java, based on delegation, but the solution have problems to deal with self calls. The solutions presented here do not have this problem. Also Malak describes in [5] a proposal to simulate multiple inheritance in Java with a solution based on delegation and implemented through Jamie. Author analyzes the problems with virtual self calls, constructors and finalizers and propose a new solution similar to the Variant 2 of the Java case. But the solution is just sketched and author states that it is not implemented. We have implemented the Variant 2 of the Java case as a preprocessor that generate Java code from an extended Java with multiple inheritance.

## 4 Conclusions and future work

From the study of multiple inheritance designs, we can state some positive and negative points. On the good side, multiple inheritance designs are important to represent the world concepts in the design model. On the bad side, not all object oriented languages allow multiple inheritance, and even they did it, with multiple inheritance appear some conflicts and we desire certain flexibility in the way they can be solved. Hence, it is desirable first, to promote multiple inheritance design, second, to have available a method for transforming multiple inheritance hierarchies into single inheritance hierarchies. And third, introduce the possibility of versatile conflict resolution in the translation.

Translation methods are not just useful for implementing multiple inheritance designs in single inheritance (or non-inheritance) languages. Translation is also useful for extending single inheritance languages with multiple inheritance and to translate from multiple inheritance languages to single inheritance languages.

We think that implementing the translation for extending single inheritance languages with multiple inheritance can be used as intermediate result in the other two cases.

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. These 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.

Experimental works we developed take into account other important aspects like creation methods, access control and abstract classes that are not presented here for reasons of brevity. The implementation is a preprocessor that generate Java code from an extended Java with multiple inheritance.

An immediate work to be developed consists in the analysis of the strategies according to how they fixed to the type of model that is wanted to transform. We must implements more strategies in order to be able of build a benchmark for numerically comparing results.

## References

1. L. Cardelli. A semantics of multiple inheritance. *Semantics of Data Types, LNCS 173*, pages 51–68, 1984. also in *Information and Computation* 76, 1988.
2. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–523, Dec 1985.
3. R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. *ACM SIGPLAN notices*, 29(10), Oct 1994.
4. F.J. García, J.M. Marqués, and J.M. Maudes. Mecanos as basis of compositional/generative mixed reuse model. In *Proceedings of II European Reuse Workshop, Madrid, Spain*, November 1998.
5. M. Malak. Simulating multiple inheritance. *Journal of Object-Oriented Programming (JOOP)*, pages 3–6, April 2001.
6. J.M. Marqués. *Jerarquías de herencia en el diseño de software orientado al objeto*. PhD thesis, Universidad de Valladolid, 1995.
7. B. Meyer. Tools for a new culture: Lessons from the design of the Eiffel libraries. *CACM*, 33(9):68–88, Sept 1990.
8. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
9. C. Oussalah, M. Magnan, and L. Torres. Multiple inheritance, you say? In *ECOOP'92 "Workshop Multiple Inheritance and Multiple Subtyping"*, 1992.
10. 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.
11. 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.

12. J.J. Rodríguez, Y. Crespo, and J.M. Marqués. Transformación de jerarquías de herencia múltiple en jerarquías de herencia sencilla. Technical Report DI-2000-0002 (TR-GIRO-03-98), Departamento de Informática, Universidad de Valladolid, 1998.
13. L. Stein. Delegation is inheritance. In *Proceedings of OOPSLA'87*, 1987.
14. E. Tempero and R. Biddle. Simulating multiple inheritance in java. *The Journal of Systems and Software*, 2000(55):87–100, 2000.
15. J. Viega, B. Tutt, and R. Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, University of Virginia, 1998. <http://www.list.org/jamie>.
16. P. Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA'87*, 1987.
17. R. Wieringa, W. de Jonge, and P. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems.*, 1(1):61–83, 1995.