# Using both Specialisation and Generalisation in a Programming Language: Why and How?

Pierre Crescenzo, Philippe Lahire

▶ To cite this version:

Pierre Crescenzo, Philippe Lahire. Using both Specialisation and Generalisation in a Programming Language: Why and How?. Workshop Managing Specialization/Generalization Hierarchies lors de la conférence OOIS 2002 (8th International Conference on Object-Oriented Information Systems), Sep 2002, Montpellier, France. pp.64-73. hal-01304663

HAL Id: hal-01304663
https://hal.archives-ouvertes.fr/hal-01304663

Submitted on 20 Apr 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.

# Using both Specialisation and Generalisation in a Programming Language: Why and How?

Pierre Crescenzo and Philippe Lahire

Laboratoire I3S (UNSA/CNRS), Projet OCL 2000, route des lucioles, Les Algorithmes, Btiment Euclide B BP 121 F-06903 Sophia-Antipolis CEDEX, France
{Pierre.Crescenzo, Philippe.Lahire}@unice.fr
http://www.i3s.unice.fr/{ crescenz/, lahire/}

**Abstract.** The reuse of libraries of classes by client applications is an interesting issue quite difficult to achieve, especially when modification of the class tree is needed but not possible because of the context. We propose a solution which is based on the presence of both specialisation and generalisation relationships in an object-oriented programming language. The specification of both relationships is based on a meta-model called *OFL* which provides a support for describing the operational semantics of a language through the definition of parameters and semantical actions. We propose an overview of the expressiveness of OFL and of its implementation and we give also some other interesting applications.

## 1 Introduction

In this paper we address the problem of the reuse of libraries of classes by client applications when modifications of the class tree is needed. We propose a solution which is based on the introduction of both specialisation and generalisation relationships in future object-oriented programming langages. This idea to combine both relationships altogether is also pointed out in [5] which focuses more on the integration feasibility within existing OOPL. According to the handling of libraries of classes there are other problems to solve like the maintenance of classes (removal of deprecated features, redefinitions, etc.) that may be solved using interclassing [4]. Even if those approaches deal with the use of libraries of classes by client applications, the philosophy is quite different: our approach deals with existing libraries that may not be modified by client applications whereas the other approach is related to the modification of libraries of classes themselves and their consequences in client applications.

To develop this idea, we present two main parts. Firstly, in section 2, we describe a very pragmatic situation where specialisation and generalisation are useful in the graph of types. You will see that the use of only one of them would lead to only a poor approximation.

Secondly, in section 3, we present a practical solution to define a new programming language with both specialisation and generalisation, or to improve an existing language with a reverse inheritance. This solution is based on the *OFL*

Model ("Open Flexible Languages") [2]. The section 4 presents some implementation issues which handle principles of the *OFL* Model. Then, we conclude the paper in the last section, 5.

## 2 Why both Specialisation and Generalisation?

Our approach is defined in the context where a programmer uses a software library of components (these components could be classes). He may have written this library or not, but he cannot modify it. This situation happens very often, for instance when the code is not provided, when it is *copyright*ed, when it has to be left unchanged for existing applications, and so on. The figure 1 give an example of such a very simple library with two very typical classes.

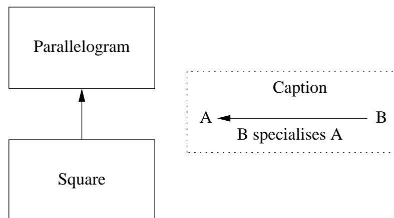


**Fig. 1.** An existing and unmodifiable hierarchy of classes

Now, for a specific program need or to make the library evolve, we want to add a component in the library (*i. e.* a class in the graph). This fact is illustrated in figure 2.

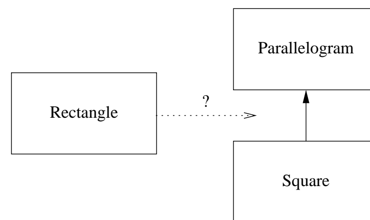


**Fig. 2.** A new class in the unmodifiable hierarchy

We can imagine three solutions to integrate Rectangle in the hierarchy:

1. The first is the most simple: "If we want to add a class, we must reorganise all the hierarchy!" This solution, illustrated in figure 3, is obviously the best one. But the best one if we can modify the hierarchy and an impossible

one otherwise. And even if we could modify the existing classes, it could be a bad idea: we could add some bugs in some other applications which use these existing library and, in the example, the stability of Square is called into doubt by introducing Rectangle.

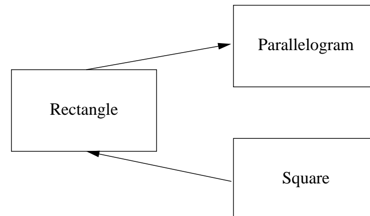


**Fig. 3.** The first solution: a total reorganisation of the hierarchy

2. A second solution respects the constraint of the unmodifiable existing hierarchy. The idea is to insert Rectangle as a specialisation of Parallelogram, as you can see in figure 4. Here there is no problem with existing classes and the relationship between Parallelogram and Rectangle is correct. But the instances of Square logically have to be instances of Rectangle and this is not the case here.

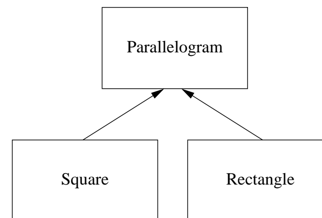


**Fig. 4.** The second solution: Rectangle specialises Parallelogram

3. The third solution is to take advantage of the fact that Rectangle is closer from Square than from Parallelogram. So, the idea is to specialise Square rather than Parallelogram as it is shown in figure 5. This solution is valid as long as polymorphism capabilities between Square and Rectangle are not used. The instances of Square logically have to be instances of Rectangle and this is the contrary here.

As we just see, if we have only specialisation (the problem is the same if we have only generalisation), we can make evolution of a graph of classes without risk (e.g. without modifying existing classes) but we can't have, simultaneously, a valid behaviour of the types (e.g. correct polymorphism capabilities) in the resulting graph.

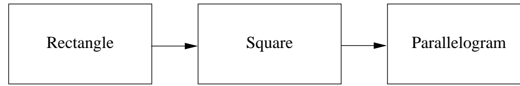


**Fig. 5.** The third solution: Rectangle specialises Square

Our proposition is to add a generalisation relationship in order to have both specialisation and generalisation in the same language. Generalisation is only the reverse link of specialisation so, theoretically, only one of them is sufficient. But practically, we could perfectly resolve our evolution problem with both. Let's show you a new figure, 6. It demonstrates a good way to handle evolution in our graph of classes. Rectangle is integrated as a specialisation of Parallelogram and a generalisation of Square. But what are the advantages in comparison with the three previous solutions?

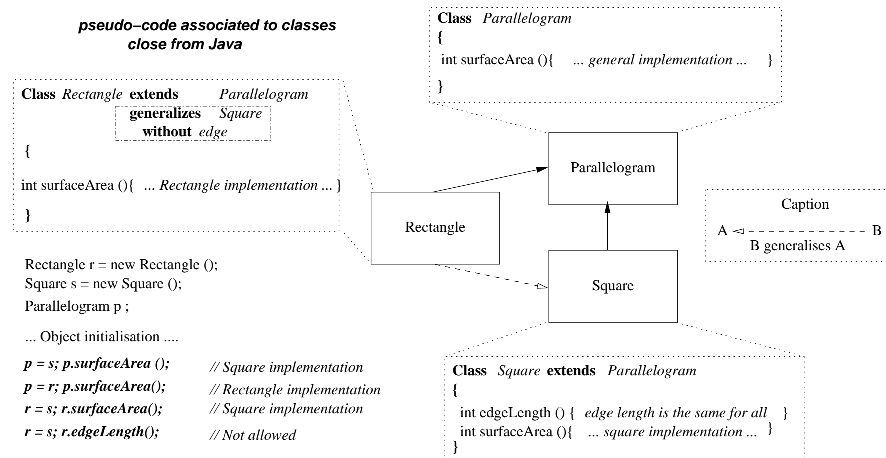


**Fig. 6.** A satisfactory integration of Rectangle with possible pseudo-code

The advantage of our solution in relation to the first one is that no class is modified in the initial graph. If we haven't the code or the right to modify it, we can nevertheless apply a relevant adaptation of the graph. And even if we can modify the initial graph, our solution protects the quality of Square since the new Rectangle must be compatible with the well-tried Square and not the contrary. In comparison with the second proposition, to use both links allows to make capital out of a correct behaviour of polymorphism between Square and Rectangle. In relation with the third approximative solution, the idea to use generalisation is better because the graph of types is relevant : a square is a rectangle and not the contrary! Obviously the pseudo-code inserted in figure 6 is not self-sufficient to explain the semantics and it should be deeply discussed.

Particularly, he has to be linked to the possible parameter values presented in 3.2 and should be further specified by assertions built on the model reification. However, in order to give a flavour of the capabilities which may be provided to programmers, we could say that in a *Java*-like language, the two keywords *extends* and *generalizes* are strongly related to lookup operation (see 3.3) in order to implement polymorphism and to allow to access to class instances as if the class hierarchy was the one described in figure 3.

# 3 How? The Model *Open Flexible Languages*

## 3.1 *OFL* in a nutshell

This section presents the *OFL* model (Open Flexible Languages) and its capability to define easily both specialisation and generalisation.

The *OFL* Model aims to describe the main object-oriented programming languages (such as *Java*, *C++*, *Eiffel*, ...) to allow their evolution and their adaptation to specific programmer's needs. To reach this goal, *OFL* reifies all elements of an object-oriented programming language in a set of components of a language. Thus classes, methods, expressions, messages, and so on are the *OFL*-components and are integrated in a specific MOP (Meta-Object Protocol) which is self-extendable and contains the set of entities needed for the reification of both languages and user applications.

The meta-programmer creates a language by selecting adequate *OFL*-components in predefined libraries. He can also specialise a given *OFL*-component in order to generate one dedicated to some specific uses.

Classes are reified by *OFL*-components. Take the example of *Java*. We have ComponentJavaClass, ComponentJavaInterface, ComponentJavaArray, ... An originality of *OFL* is that relationships are also reified. So, we have for *Java*: ComponentJavaExtendsBetweenClasses, ComponentJavaImplements, etc[1].

To facilitate the creation of an *OFL*-component, *OFL* provides some meta-components, called *OFL*-concepts. So, we have a ConceptRelationship and a ConceptDescription (the word *description* has been chosen to represent classes and all entities which look like classes, such as interfaces). Thus, ConceptDescription as well as ConceptRelationship are equivalent to meta-meta-classes. In each concept, a set of parameters gives the meta-programmer necessary expressiveness to create or adapt an *OFL*-component.

## 3.2 Hyper-Generic Parameters

But how can the meta-programmer easily define the *OFL*-components for the language he wants to create or adapt? In fact, this work may be very difficult and tedious because he would have to rewrite a lot of algorithms such as type controls, dynamic links, use-of-polymorphism verifications, inheritance rules, etc..

[1] The full list of *OFL*-components for *Java* is given in [1].

In *OFL*, we provide a way to simplify this task: hyper-generic parameters. All the algorithms are predefined and are customized by hyper-generic parameters which have a value in each *OFL*-components.

In the sequel, we illustrate a subset of the hyper-generic parameters which can be applied to an *OFL*-component reifying a relationship to customize it. We explain each parameter and its capabilities of customization, and we give its value when it is mandatory for the definition of ComponentSpecialisation and ComponentGeneralisation.

**Kind** In *OFL*, we handle four kinds of relationships: import for inheritance and all other importation links between descriptions, use for aggregation, composition, and all other use links between descriptions, type-object for all links between types and objects such as instantiation, and objects for all links between objects. Its value for ComponentSpecialisation and ComponentGeneralisation is import.

**Cardinality** It defines the maximal cardinality of a relationship. For example, the value of Cardinality is $1-1$ for a single inheritance and $1-\infty$ for a multiple one. We want to specify single links, so its value is $1-1$ for ComponentSpecialisation and ComponentGeneralisation.

**Repetition** It is useful if and only if Cardinality is not $1-1$ (to implement repeated inheritance, for example). For ComponentSpecialisation and ComponentGeneralisation, the value of Repetition is ignored.

**Circularity** It expresses if the *OFL*-component admits a circular graph (it is useful mainly for *use* relationships). Circularity is forbidden for ComponentSpecialisation and ComponentGeneralisation.

**Symmetry** This parameter points out if the *OFL*-component provides relationships that are symmetrical (e.g. a is-a-kind-of relationship). Neither ComponentSpecialisation nor ComponentGeneralisation is symmetrical.

**Opposite** We may have, in a language, two *OFL*-components with reversed semantics. This is an essential information for all actions which need to navigate through the graph of descriptions (e.g. to ensure type conformance). ComponentSpecialisation and ComponentGeneralisation are opposites.

**Direct_access and Indirect_access** These parameters give the capability to choose the policy of this visibility. In traditional inheritance, features of the ancestor are directly visible in the heir, as if they are declared in it. Some languages propose also to name the target-description for example to access to the old-version of a redefined method. For ComponentSpecialisation and ComponentGeneralisation it may be allowed or not[2].

**Polymorphism_implication** It can take four values: up means that all instances of the source-description must be also instances of the target-description; down points out the contrary (very useful to specify a generalisation); both means that source-description and target-description have the same instances (it is useful to describe versionning); none allows for example to define relation-

[2] All languages may not provide the same expressiveness. Just think about the difference about the handling of inheritance in *Java*, *C++*, *Eiffel* or *Sather*, etc.

ships dedicated to code reuse. The value for ComponentSpecialisation is up and it is down for ComponentGeneralisation.

**Polymorphism_policy** It indicates if a new declaration of attribute or method in the source-description hides the feature in target-description or overrides it. For ComponentSpecialisation and ComponentGeneralisation, we can use a traditional value: hiding for attributes and overriding for methods[3].

**Feature_variance** It proposes three kinds of variance rule for the redefinition of features : covariant like in *Eiffel* (the type indicated in the source-description must be the same or a subtype according to Polymorphism_implication), contravariant as in *Sather* [6] (this is the reverse), nonvariant as in *Java*[4] (the type indicated in the source-description must be the same than the one in the target-description). ComponentSpecialisation and ComponentGeneralisation do not impose a specific value but their status of opposite means a coordinated choice.

**Assertion_variance** It takes into account languages with assertions like *Eiffel*. It indicates the kind of variance for assertions: weakened (the assertion of source-description must be implicated by the assertion of target-description), strengthened (this is the reverse), unchanged (they must be equivalent).

**Renaming, Adding, ...** The First one is dealing with the right to rename a feature through a relationship defined by the *OFL*-component[5]. *OFL* also provides parameters to customize the capability to add, to remove, or to redefine assertions, method's signatures, method's bodies, and method's qualifiers, but also to mask, to show, to abstract, or to make effective the imported features. For example, according to ComponentSpecialisation, it is relevant to add a feature to a specialised class but not to remove any of them. To redefine a feature is also possible. ComponentGeneralisation should have the opposite semantics : for instance we should be able to remove but not add some feature.

### 3.3 Actions

To associate values to a set of parameters may be appropriate for describing the customized behaviour of a sort of relationship. But we need more to allow relevant control and execution of these links so that *OFL* includes a list of actions.

Each action defines the operational semantics of a part of work traditionally handled during the compilation or execution time. And each action takes into account the value of the hyper-generic parameters. So the behaviour of the defined language is adapted to the value of each parameter of each component. We have classified our actions in seven categories: **actions to search a feature** (e.g. lookup to find the relevant feature in the graph of descriptions according to a message), **actions to execute a feature** such as execute which allows to perform a routine call, **actions to make a control** like are_valid_parameters which

[3] In *OFL*, overloading is not handled by relationships but by descriptions.
[4] If type of parameters of methods are not exactly the same, in *Java* this is overloading and not overriding.
[5] Renaming is possible in *Eiffel* but not in *Java* or *C++*.

controls the compatibility between effective and formal parameters, **actions to handle instances of descriptions** (e.g. create_instance or destroy_instance), **actions to handle extension of descriptions** or **Base operation** such as assign or copy.

**How to write an action?** An action could be simple such as verify_circularity which controls that all relationships with the parameter Circularity not set don't make circular graph. The algorithm of this action is simple: to go all over the graph for this relationship and to verify that none of the descriptions is direct or indirect target of itself. The moment to execute verify_circularity is also easy to imagine: it could be launch once in a static tool like a compiler or a code checker.

But other actions are a lot intricate! For example, let's examine the action lookup. To find the relevant feature in accordance with a message, the task may be difficult and the algorithm complicated. We have to take the value of many parameters into account. The value of Polymorphism_implication is used to build a graph of types. Polymorphism_implication will help to determine which policy (hiding or overriding) have to be considered. With Cardinality and Circularity, we can choose an efficient way to go all over the graph. Symmetry could help us to adopt a two-direction route. Direct_access and Indirect_access give information about the visibility of the target-description. Finally, Feature_variance, Assertion_variance, Adding, Removing, and so on, allow to know how features are imported. Furthermore, the moment when it is correct to execute the lookup is also not obvious. We can easily imagine that a first part of this task is static (determination of all unambiguous calls for example) and another one is dynamic (dynamic linkage at runtime for example).

Then we may understand that it is possible to write the code of lookup. But if we want to provide some useful model to the programmer, it is obviously necessary to help him to write actions. In this way, we supply three things. The first one is that we have split complex actions in more elementary ones[6]. For example, we have a local_lookup which make the local (independently of all import relationships) research of a relevant feature in a description and a match which takes a feature and a message and determines if the second one is compatible with the first one... Thus, we split the difficulty of the complex lookup which has to call local_lookup, match and other actions to make its job.

Secondly, to solve the problem of the static and dynamic facets of our actions, we provide a way to define them in several parts. So, in fact, each action is split in a set of facets and each facets is declared static (used in a preliminary step like a compiler, a code checker, or a first access) or dynamic (used in an interpreter, an execution engine, or a virtual machine).

Thirdly, we intend to provide some default behaviours for all actions. Indeed, *OFL* could be used for a large variety of tools about source code. In this paper, we present a way to assist an extension of a programming language, but it is also possible to use our actions to make others tools like a code checker, some trace

[6] Chapter 6 of [2] presents more than fifty actions.

service, or a wizard for programming. So, our idea is to write typical algorithms for actions and to supply them in libraries.

**Who write the actions?** There are three possible answers to this question. As we just explained, *OFL*-designers (we) have to provide libraries of actions for the more frequent usages. These libraries must be for very general purposes. When a relevant solution is not given in these libraries, the meta-programmer (the person who designs a language or a tool for handling source-code) has to redefine some of the actions or, in a bad case, to rewrite all of them. It is here useful to create a kind of plug-in library which adds some interest to the *OFL* set of tools. Finally, when the meta-programmer wants to add a very particular behaviour, he can redefine or write some actions in order to handle this behaviour. As this case is for a specific use (useful for an unique application, for example), creation of a library is not useful and the redefinition could be temporary.

## 4 Implementation issues

Firstly, an implementation of *OFL* (cf. fig. 7) is based on the reification of both language semantics (*OFL*-components instances of an *OFL*-concepts) and application entities such as method, attribute, statement, etc. Because it is not reasonable to design a reification which deal with any entity of any language, it is necessary to design an extensible reification model. All this issues are achieved through a Meta-Object Protocol (MOP) written in *Java* and called *OFL/J*. In order to make easier the coupling with other tools, an XML-DTD of *OFL/J* can be generated automatically whereas meta-information and application reification are stored under an XML representation which conforms to this DTD.

Secondly, the reification of application should be parsed and semantics actions should be performed on each entities according to the language semantics. This will be done by *SmartTools* [3] which allows to define visitors (design-pattern) in order to allow the description of semantical actions to be associated to application entities. *SmartTools* apply all these actions, automatically to any node of the abstract syntax tree associated to the application reification.

One interesting thing is the flexibility of the system. Actions can be added or removed from *OFL/J* and they can implement the approach described above from different point of view: to control the appropriateness between the body of application methods and the relationships defined between the classes within the reification, to generate pure *Java* code according to the information above, or to do both control and generation. Many other variants may be found according to the level of reification of statements and expressions (e.g. to insert into action-semantics the code for implementing an open virtual machine).

## 5 Conclusion and future work

In this paper we demonstrated that it was interesting to make coexist both specialisation and generalisation relationships, in order to better handle libraries

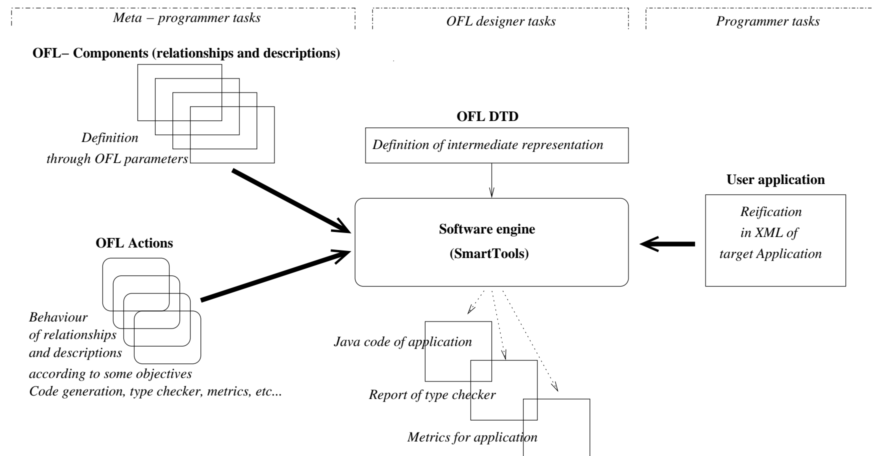


**Fig. 7.** Architecture of OFL implementation

of classes in the design of application. Other relationships also could be useful such as a reuse-code relationship whose aim is to provide one class the capability to include some methods from existing classes without allowing any polymorphism for its instances with those classes. These are only examples of the kind of relationships that *OFL/J*, the implementation of *OFL* model could handle. the part of *OFL/J* which deals with meta and non meta information reification and with the *OFL* Mop for extending the capabilities of the reification are implemented. Now we are investigating how to implement a first version of the semantics actions into *SmartTools*.

## References

1. A. Capouillez, P. Crescenzo, and P. Lahire. Le modele OFL au service du meta-programmeur - Application Java. In *LMO'2002*. Hermes Sc Pub., *L'objet*, vol. 8, N 1-2/2002, Jan. 2002.
2. P. Crescenzo. *OFL : un modele pour parametrer la semantique operationnelle des langages a objets - Application aux relations inter-classes.* PhD. Thesis, University of Nice-Sophia Antipolis, December 2001.
3. D. Parigot. Web Site of *SmartTools*. World Wild Web, Dec. 2001. http://www-sop.inria.fr/oasis/SmartTools/.
4. P. Rapicault and A. Napoli. Evolution d'une hierarchie de classes par interclassement. In *LMO'2001*. Hermes Sc. Pub., *L'objet*, vol. 7, N 1-2/2001, jan. 2001.
5. M. Sakkinen. Exheritance - Class Generalisation Revived. In *ECOOP'2002 (The Inheritance Workshop)*, jun. 2002.
6. D. Stoutamire and S. Omohundro. Sather Specification. Technical report, International Computer Science Institute, University of Berkeley, Aug. 1996.