

Customisation of Inheritance

Pierre Crescenzo, Philippe Lahire

▶ **To cite this version:**

Pierre Crescenzo, Philippe Lahire. Customisation of Inheritance. The Inheritance Workshop lors de la conférence ECOOP 2002 (16th European Conference on Object-Oriented Programming), Jun 2002, Malaga, Spain. pp.23-29 et 117-134. hal-01306686

HAL Id: hal-01306686
https://hal.science/hal-01306686

Submitted on 25 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.

CUSTOMISATION OF INHERITANCE

Pierre Crescenzo and Philippe Lahire

Address: Laboratoire I3S (UNSA/CNRS)
Projet OCL
2000, route des lucioles
Les Algorithmes, Bâtiment Euclide B
BP 121
F-06903 Sophia-Antipolis CEDEX
France
E-Mails: Pierre.Crescenzo@unice.fr
Philippe.Lahire@unice.fr
Web: http://www.crescenzo.nom.fr/
http://www-iutinfo.unice.fr/~lahire/

Keywords: Customised Relationships, Meta-Programming, Hyper-Generic Parameters
Abstract: This paper presents the model *OFL* with its capabilities to describe a set of variability criteria for relationships between classes in general, and inheritance in particular. Our main goal is to show pragmatic criteria which can be used to define or customise the behaviour of inheritance.

1 Introduction

Inheritance is an essential and powerful concept of object-oriented languages. It is applied to a great number of different problems [Mey97] such as type specialisation, reuse of pieces of code, class versioning, type generalisation [CCCL00], ...

However, inheritance is not a panacea: while it can be used for a large range of programmer's needs, the excessive use of inheritance represents also some disadvantages [LJ95], e. g. loss in code readability, code reliability or code evolution. Those losses may be due, for example, to a non-relevant use of polymorphism when the need is only to reuse a piece of code or to a reverse use of polymorphism when type generalisation is concerned. Moreover, making a new class version through the use of inheritance leads a loss of capability with respect to feature adaptation (for example, inheritance does not allow feature removal).

This paper presents some elements of a solution to these problems. This solution is based on the model *OFL* [CCL01,Cre01,CCL02] which is presented in section 2. Section 3 on the next page focuses on the notion of hyper-generic [Des94] parameters which allow to define the inheritance relationship, and to customise it. Section 4 on page 6 propose a brief overview of the software tools and last section, 5 on page 6, concludes the paper and presents future works.

2 Open Flexible Languages

This section presents the model *OFL* (Open Flexible Languages).

OFL is a model to describe the main object-oriented programming languages (such as *Java* [GJSB00], *C++* [Str97], *Eiffel* [Mey92], ...) 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 allows to extend 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. (S)he can also specialise a given *OFL*-components in order to generate one dedicated to some specific uses. To separate the default *OFL*-components of the *OFL*-components created for a specific language, we call *OFL*-atom the default one.[1]

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, ComponentJavaExtendsBetweenInterfaces, ComponentJavaImplements, ... A more complete list of *OFL*-components for *Java* is given in [CCL02].

To facilitate the creation of an *OFL*-component, *OFL* provides some meta-components, called *OFL*-concepts. So, we have a ConceptRelationship and a ConceptDescription[2]. Thus, ConceptDescription is equivalent to a meta-meta-class. In each concept, a set of *parameters* gives the meta-programmer powerful possibilities to create or adapt an *OFL*-component. These parameters are detailed in section 3.

Each parameter corresponds to a part of the operational semantics of the described *OFL*-components. To execute operations in accordance with these parameters, *OFL* provide a system of *actions*. You can see an action as a part of a compiler or of a running engine. Here are two examples of action:

lookup This action searches for the relevant feature in conformity with a message. This is the realisation of the dynamic link. lookup takes into account the value of parameters which handle polymorphism, redefinition and variance for each encountered relationship.

is_conform_type The goal of this action is to verify if a type is conform to another one. Conformity is defined by parameters about polymorphism of import relationships (like inheritance).

In order to sum up this short definition of *OFL*, we present the figure 1 on the next page where the model is summarised and some *OFL*-components for *Java* are defined.

3 Hyper-Generic Parameters

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

In *OFL*, we provide a way to simplify this task: hyper-generic parameters. All the algorithms are predefined (that is to say that all action have a default algorithm which takes care of the value of hyper-generic parameters, and meta-programmer can redefine these actions) and are customised by hyper-generic parameters which have a value in each *OFL*-components.

In the sequel, we illustrate the set of hyper-generic parameters which can be applied to an inheritance *OFL*-component to define it. We explain each parameter and its capabilities of customisation, and as an example, we give its value to define ComponentJavaExtendsBetweenClasses.

Name This is the most simple hyper-generic parameter. It is the name of the *OFL*-component and it must be unique in a language. For ComponentJavaExtendsBetweenClasses, the name is "ComponentJavaExtendsBetweenClasses".

Kind It allows to determine the sort of the *OFL*-component. In *OFL*, we have four kinds of relationships:

[1] In other words, *OFL*-atoms are supplied by the model, other *OFL*-components, created for a specific language, are not.

[2] The word *description* has been chosen to represent classes and all entities which look like classes, such as interfaces.

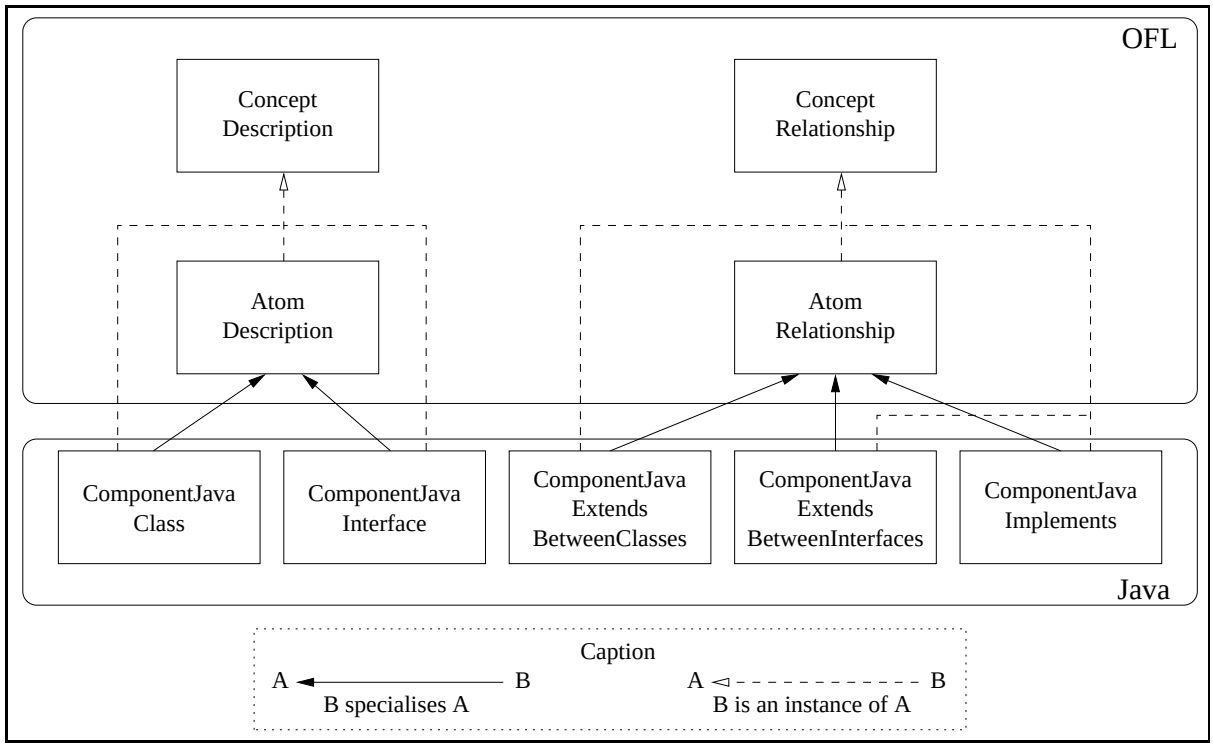


Fig. 1. *OFL* and *Java* described by *OFL*

- 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.

The value of Kind for ComponentJavaExtendsBetweenClasses is obviously import.

Context This is a simple but useful parameter. Context is used to know if the *OFL*-component is defined for a specific language (value: language) or in a very general way and included in a library (value: library). This is important because some other parameters, such as Opposite, as you will see later, cannot be defined if the *OFL*-component is not described in the context of a language. ComponentJavaExtendsBetweenClasses is defined for *Java*, so here the value of Context is language.

Cardinality The parameter Cardinality 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. The first number represents the number of source-descriptions (heirs), the second is the number of target-descriptions (ancestors).[3] So, with Cardinality, we can customise the relationship to be single or multiple with a single value! All the difficulty of the lookup algorithm, which searches the relevant method in the graph of descriptions, is encapsulated in a predefined action which takes care of the Cardinality value for all relationships used in the application. Cardinality is also useful to limit the multiplicity. Indeed, giving the value $1-3$, you can limit your multiple inheritance to have one, two, or three ancestors, and not more. In *Java*, inheritance between classes is single, so Cardinality for ComponentJavaExtendsBetweenClasses has the value $1-1$. If we take ComponentJavaExtendsBetweenInterfaces, we have $1-\infty$.

[3] All the *OFL*-components we have defined with *OFL* have the value $1-1$ or $1-\infty$ for Cardinality. But we keep the capability to make a $\infty-\infty$ *OFL*-component to represent, for example, association of *UML* [Obj01].

Repetition This parameter is useful if and only if Cardinality is not $1-1$. Repetition indicates if repetition of source-descriptions and target-descriptions are valid for this *OFL*-component (to make repeated inheritance, for example). Repetition is defined as a pair of boolean. For ComponentJavaExtendsBetweenClasses, the value of Cardinality is $1-1$, so the value of Repetition is ignored.

Circularity This is a boolean and it expresses if the *OFL*-component admits a circular graph (value: true) or not (value: false). Often, *use* relationships allow circularity and *import* ones don't. Circularity is forbidden in inheritance of *Java*, so the value of Circularity is false for ComponentJavaExtendsBetweenClasses.

Symmetry This parameter points out if the *OFL*-component provides relationships that are symmetrical. Most of traditional links are not, but we can imagine a ComponentIsAKindOf where the semantics is bidirectional: a boat is-a-kind-of submarine and a submarine is-a-kind-of boat (they resemble each other but none are a specialisation of the other). ComponentJavaExtendsBetweenClasses is not symmetrical so the value for its Symmetry is false.

Opposite We may have, in a language, two *OFL*-components with reversed semantics. Let's imagine ComponentSpecialisation and ComponentGeneralisation. Each indicates the other as its opposite. This is an essential information for all actions which need to travel through the graph of descriptions. ComponentJavaExtendsBetweenClasses has no opposite, so the value of this parameter is none.

Direct_access In traditional inheritance, features of the ancestor are directly visible in the heir, as if they are declared in the heir. The parameter Direct_access gives the capability to choose the policy of this visibility. If the value is mandatory then all features are inevitably visible. If it is forbidden, none are directly visible (but they can be indirectly visible as we will see in the next parameter). And if the value is allowed then some are visible, some not and the differentiation may be done, for example, by a keyword (such as public, private, ...). For ComponentJavaExtendsBetweenClasses, the relevant value is allowed.

Indirect_access This is the same idea as for the previous parameter but for indirect accesses. Indirect accesses mean accesses naming the target-description. In *Java*, we can use super in constructors, finalisers or redefined methods. By this way, we can access to some features of the ancestor, but we have to specify an indirect access. So, for ComponentJavaExtendsBetweenClasses, the value is allowed.

Polymorphism_implication This parameter is very important. Polymorphism_implication can take four values:

- up means that all instances of the source-description (heir in an inheritance link) must be also instances of the target-description (ancestor in an inheritance link). This is the traditional direction for polymorphism.
- down points out the contrary: all instances of the target-description must be also instances of the source-description. This value is very useful to create *OFL*-components like ComponentGeneralisation.
- both is an interesting value. It means that source-description and target-description have the same instances. This can be relevant to describe other derivations of inheritance, such as ComponentVersion. We can imagine two versions of class linked by this *OFL*-component. The two versions represent the same type, so they must have the same list of instances, and dynamic link has to find the good version of features to execute.
- none is the last possible value and allows to define other kinds of inheritance, such as ComponentCodeReuse where features are imported from the target-description to the source-description, but we need to ensure that polymorphism capabilities are avoided.

The value of Polymorphism_implication for ComponentJavaExtendsBetweenClasses is up.

Polymorphism_policy This parameter is ignored if Polymorphism_implication has the value none. Polymorphism_policy indicates if a new declaration of a feature in the source-description hides the feature in target-description (value: hiding) or redefines it (value: overriding). This value is double, one for attributes, one for methods. For ComponentJavaExtendsBetweenClasses, the value is hiding for attributes and overriding for methods.[4]

Feature_variance This parameter indicates the kind of variance rule for redefinitions of features, if these redefinitions are allowed (we will see the parameter Redefining later). Four values are possible:

- covariant The type indicated in the source-description must be the same or a subtype[5] of the type given in the target-description. This is the relevant value for the parameters of methods of *Eiffel*.
- contravariant This is the reverse of the previous value. The type indicated in the target-description must be the same or a subtype of the type given in the target-description. This choice has been made, for example, by *Sather* [SO96].
- nonvariant The type indicated in the source-description must be the same than the type given in the target-description. This is the case in *Java*[6].
- non_applicable is the last possible value. Meta-programmer uses it if (s)he wants no feature-variance control.

The value of Feature_variance for ComponentJavaExtendsBetweenClasses is nonvariant for method parameters, nonvariant for function results, and non_applicable for attributes.

Assertion_variance *OFL* is able to describe languages with assertions (precondition, postcondition, and invariant) like *Eiffel*. So, we have a parameter to indicate the kind of variance for assertions:

- weakened The assertion in the source-description must be implicated by the assertion in the target-description.
- strengthened This is the reverse value of the previous one. The assertion in the source-description must implicate the one of target-description.
- unchanged The assertions in source-description and target-description must be equivalent.
- non_applicable means that controls of assertion variance must be avoided.

For ComponentJavaExtendsBetweenClasses, the value of Assertion_variance is ignored, because *Java* has no precondition, postcondition, or invariant.[7]

Renaming This parameter points out if the programmer can rename a feature using a relationship defined by the *OFL*-component. For example, renaming is possible in *Eiffel* but not in *Java* or *C++*. The accepted values are forbidden to prevent renaming, allowed to authorise renaming, or mandatory to oblige it. The value for ComponentJavaExtendsBetweenClasses is forbidden.

On the same idea than for Renaming, we have parameters to customise the capability to add (Adding), to remove (Removing), to redefine (Redefining for assertions, method's signatures, method's bodies, and method's qualifiers), to mask (Masking), to show (Showing), to abstract (Abstracting), or to make effective (Effecting) the imported features. The value for ComponentJavaExtendsBetweenClasses is allowed for Adding and Redefining (only for method's bodies and method's qualifiers) and forbidden for all others.[8]

[4] In *OFL*, capabilities of overloading is not handle by relationships but by descriptions.

[5] Let A be the source and B the target. A is a subtype of B if the value of Polymorphism_implication is up, and B is a subtype of A if Polymorphism_implication is down. If the value is both, A and B represent the same type and if it is none, there is no subtype link between A and B.

[6] If type of parameters of methods are not exactly the same, in *Java* this is overloading and not overriding.

[7] A keyword assert is present in *Java* 1.4.0 to handle assertions but this is a very basic *ad hoc* mechanism.

[8] extends between an abstract class and a concrete one is handled by another *OFL*-component.

All these hyper-generic parameters allow to easily create many different kinds of inheritance, and to directly execute them almost without meta-programming. And we need to write meta-programming code only if we want to modify or to advance the default semantics of actions which take care of the values of the hyper-generic parameters[9].

4 Tools

The *OFL* model is defined since December 2001 and we are now implementing several software tools. The first one is a *Java* version of the model which reifies all *OFL*-atoms (the programming language elements such as method, description, message, ...) and *OFL*-concepts (the meta-components) and provides an *OFL*-MOP with hyper-generic parameters and actions. This *Java* library is called *OFL/J* and is also equipped by capabilities to save and load all entities conforming to an *XML-Schema* [Wor01]. A full documented release should be soon available on our Web sites. Without these tools, *OFL* is a way to classify and define components of languages, with it, it will become a platform to construct language, to test evolution to existing language, or to equip applications with controls or other behaviours.

We are also implementing some graphical tools to help *OFL* users. *OFL-Meta* will be used by the meta-programmer to create and modify the *OFL*-components of a language. Its interface resembles to the interface of the *Windows-File-Explorer*. Another tool looks like an *UML* graph editor. It is called *OFL-ML* and its goal is to provide a language (made through *OFL-Meta*) to the programmer and to give to him(her) graphical solution to make his(her) application. A syntax is not yet specified (only a reification) in the current version, so method bodies are written using the *Java* syntax. Currently, those tools are only at the stage of prototype.

5 Conclusion and future work

This paper has presented a way to customise the inheritance relationship through the *OFL* model. In the very near future, we aim to use the *OFL*-MOP implementation to address the two following issues: to build a preprocessor of *Java* in order to implement an extension of this language for the customisation of the inheritance relationship, and to build a tool which uses the reification of both language semantics and application description in order to perform semantics controls, metrics, adequate source-code generation and so on. We also currently study how to use *SmartTools* [Par01] in order to implement a prototype which addresses these issues.

References

[CCCL00] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. How to Improve Persistent Object Management using Relationship Information? In *WOON'2000 (4th International Conference "The White Object Oriented Nights")*, June 2000. also Research Report I3S/RR–2000-01–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis), `http://www.crescenzo.nom.fr/`.

[CCL01] A. Capouillez, P. Crescenzo, and P. Lahire. Separation of Concerns in OFL. In *ECOOP'2001 (Workshop Advanced Separation of Concerns)*, June 2001. also Research Report I3S/RR–2001-07–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis), `http://www.crescenzo.nom.fr/`.

[CCL02] A. Capouillez, P. Crescenzo, and P. Lahire. Le modèle OFL au service du méta-programmeur - Application à Java. In *LMO'2002 (Langages et Modèles à Objets)*. Hermes Science Publications, *L'objet : logiciels, bases de données, réseaux*, volume 8, numéro 1-2/2002, January 2002. also Research Report I3S/RR–2001-04–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis), `http://www.crescenzo.nom.fr/`.

[Cre01] P. Crescenzo. *OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes*. PhD. Thesis, Université de Nice-Sophia Antipolis, December 2001. `http://www.crescenzo.nom.fr/`.

[9] Default algorithms of actions are obvious complicated because of the combination of value of the parameters to handle.

[Des94] P. Desfray. *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Sun Microsystems Press Java Series. Sun Microsystems, June 2000. `http://java.sun.com/docs/books/jls/`.

[LJ95] P. Lahire and J.-M. Jugant. Lessons Learned with Eiffel 3: the K2 Project. In *TOOLS 95*, July-August 1995.

[Mey92] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992. `http://www.eiffel.com/doc/`.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2nd edition, 1997. `http://www.eiffel.com/doc/oosc/`.

[Obj01] Object Management Group. *Unified Modeling Language Specification (UML)*, September 2001. Version 1.4, `http://www.omg.org/technology/uml/`.

[Par01] D. Parigot. Web Site of *SmartTools*. World Wild Web, December 2001. `http://www-sop.inria.fr/oasis/SmartTools/`.

[SO96] D. Stoutamire and S. Omohundro. Sather Specification. Technical report, International Computer Science Institute, University of Berkeley, August 1996. Version 1.1, `http://www.icsi.berkeley.edu/~sather/Documentation/Specification/Sather-1.1/`.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., 3rd edition, 1997. `http://www.research.att.com/~bs/3rd.html`.

[Wor01] World Wide Web Consortium. *XML Schema*, May 2001. Version 1.1, W3C Recommendation, `http://www.w3.org/XML/Schema`.