Separation of Concerns in *OFL*

Adeline Capouillez, Pierre Crescenzo, and Philippe Lahire
Laboratoire I3S (UNSA/CNRS)
Projet OCL
Les Algorithmes, bâtiment Euclide B
2000, route des lucioles
B.P. 121
F-06903 Sophia Antipolis CEDEX
France
{Adeline.Capouillez|Pierre.Crescenzo|Philippe.Lahire}@unice.fr
http://www.i3s.unice.fr/~ocl/

1 Overview of the approach

OFL (*Open Flexible Languages*) is the name of a meta-model for object-oriented programming languages based on classes. It relies on three essential concepts of these languages: the descriptions which are a generalisation of the notion of class, the relationships such as inheritance or aggregation and the languages themselves. *OFL* provides a customisation (hyper-genericity) of these three concepts in order to adapt their operational semantics to the programmer's needs. So, at first reading the *OFL* approach can be summed up as the search for a set of parameters whose value determines the operational semantics of an object-oriented language based on classes.

One of the main goals of *OFL* is to describe existing object-oriented languages such as *Java* [Fla99, GJSG00], *Eiffel* [Mey97], or *C++* [Str97] in order to be able to extend them according to two directions. On the first hand, we want to extend the language itself by adding new concepts and more especially new inter-classes relationships (like a multiple inheritance relationship in *Java*) [CCL01]. On the other hand, we wish to extend the language with new orthogonal services (for example persistent object handling). In this paper we focus on the second direction.

When designing and programming applications, people often need additional services in order to handle object evolution, persistence, distribution of data, etc. One of the most interesting approach is to use aspect-oriented programming (AOP) in order to integrate those services [KHH+01]. This integration is performed directly within the application, and most of the time there is not only one aspect dealing, for instance, with the management of persistence but several: each handles persistent object management with its own point of view. The design of hierarchies of aspects relies on the same problems as the design of the hierarchy of classes related to the main purpose of an application.

We intend to propose another variant based on an integration of aspects within the semantics of the language and not directly into an application. We aim that the handling of one service becomes the job of a meta-programmer instead of remaining the job of the application programmer. Our approach is based on the integration of aspects into *OFL* and on the ability to customise them. So that, they can be adapted to the application needs. Such approach based on hyper-genericity encourages, from our point of view, the design of aspects which are more efficient (built by a specialist) and reusable by a wide range of application (they are customisable); it improves also the separation between the implementation of a service and the design of an application. The integration of the notion of service in *OFL* uses techniques which come from AOP but these techniques are adapted in order to match the requirements of the *OFL* model.

First we give an overview of the *OFL* model core and then we give some of the key elements of our approach.

2 The hyper-genericity

Genericity is the ability to customise the behaviour of a class in an object language just as in the *Eiffel* or *C++* (`template`) generic classes. Hyper-genericity is the ability to customise the behaviour of the language itself. More precisely we have chosen to customise the behaviours of three important notions of object languages based on classes quoted above.

2.1 The Parameters

We have defined a set of parameters [CCCL01] which represents the main features of the behaviours of these three important notions which are called *concept-relationship*, *concept-description* and *concept-language*. For instance, concerning the concept-relationship, the value of the `Cardinality` parameter allows to specify if it is simple or multiple. As for the concept-description we have for example the `Generator` parameter which determines whether the concept-description can or cannot create its own instances.

2.2 The Actions

The operational semantics of each concept must adapt to the value of its parameters. This is achieved thanks to a set of action algorithms whose execution depends on these values. For example, the assignment of an object to an attribute, the dynamic binding of the features, the sending of messages and lots of other behaviours are expressed according to parameters of concept-relationship and concept-description. *OFL* links two facets to each action: the first illustrates the static part inside an interpretor or a compiler; the second represents the dynamic aspect integrated within the runtime. The distribution of the code into these two facets depends on implementation choices of the *OFL* model.

3 Architecture of the model core

Figure 1 illustrates how to use the *OFL* model to describe an application. In this figure the three necessary levels of design are shown:

1. the application level includes the program's descriptions and objects (*OFL-instances* and *OFL-data*),
2. the language level describes the components of the programming language (*OFL-components*), and
3. the *OFL* level represents the reification of those components (*OFL-concepts* and *OFL-atoms* which are in the *OFL-core*).

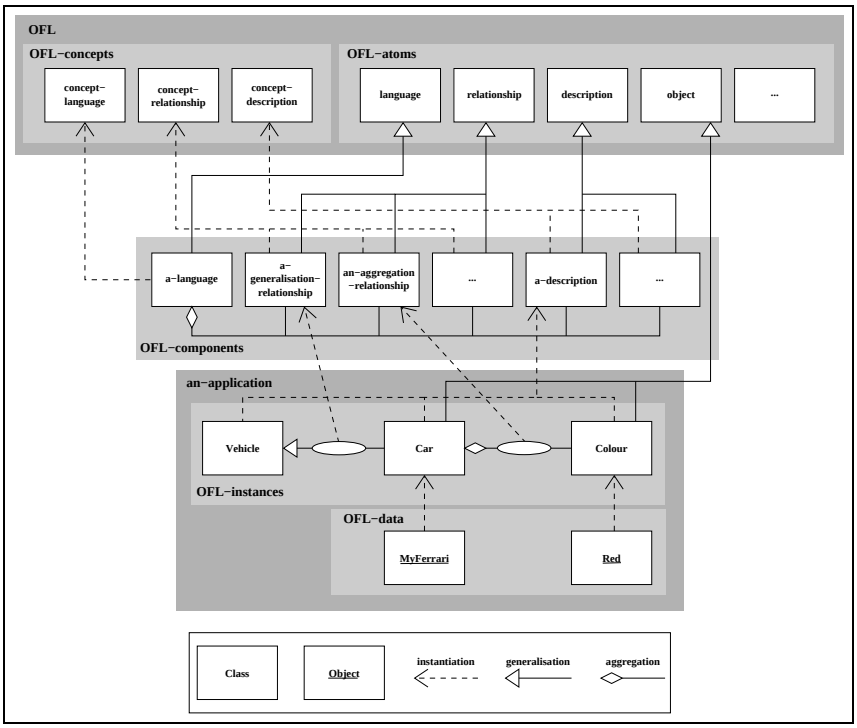

Figure 1: The *OFL*-core architecture

The Application Level To describe an application, the programmer uses the services supplied by the language level. He creates OFL-instances, which are the descriptions and the relationships of his application by instantiation of the OFL-components. At runtime, the application objects, called OFL-data, are instances of the OFL-instances representing the descriptions.

The Language Level The language level describes different types of relationships and descriptions which can be used in the described language. This entities are called OFL-components. Each OFL-component is associated to a set of values, each of them associated to one parameter of the corresponding OFL-concept. The relationships are instances of `concept-relationship`, the description are instances of `concept-description`. The language itself is an instance of `concept-language`. Its main function is to put together the relationships and descriptions which are supplied to the programmer.

The *OFL* Level This level is the core of the *OFL* model. This model is a meta-model for the programming language (language level) and a meta-meta-model for the programs (application level). As was said in section 2, we have chosen to customise three important notions: relationships, descriptions and languages. However, a lot of other entities need to be reified such as objects, methods, assertions, etc. in order to design a language completely. The *OFL* level includes two types of entities:

1. the OFL-concepts which describes the customisable part of the relationships, descriptions and languages, and
2. the OFL-atoms which reify the non-customisable part of these three concepts as well as all the other program entities (feature, instance, etc.).

Also assertions are described in each OFL-concept and OFL-atom in order to keep the model consistent, they apply on any instance of all concepts.

Let us give some more details about OFL-concepts. They are composed by several parts, main ones are:

- **Characteristics** These are the essential elements for the definition of the concept. For example, one from the description-concept is dealing with the kinds of relationship that may be used starting from associated descriptions.
- **Parameters** The value of these parameters describe a part of the operational semantics of the concept.
- **Assertions** These are the properties and constraints which are specific to all instances of one concept and to the atoms which participate to its reification.
- **Actions** They realise the operational semantics of the concept in accordance with the values of the parameters.

Characteristics, parameter values and assertions may differ from one OFL-component definition to another whereas actions are common to all OFL-components.

4 *OFL* and separation of concerns

One of the main interests of techniques related to the separation of concerns is to provide a way to integrate treatments which are orthogonal to the design of the application's hierarchy of classes, in such a way that they are clearly separated from the code of application. To build aspects which are highly reusable and adaptable is a fairly difficult thing and the skills needed for this are quite different from the ones needed for building an end-user application: generally to build one service we need an expert from the domain.

According to the *OFL* model such services could be integrated in two different ways:

- Through the definition of two additional OFL-components: a new kind of description (an aspect may be seen as a special class) and a new kind of inter-class relationship in order to simulate the *joint points* declaration [KHH+01] associated to an aspect[1].

[1] At the moment we investigate in order to check the faisibility of this implementation choice.

- To extend *OFL* with a new entity that we call OFL-aspect which will implement at the model level a way to customise OFL-concepts in order to integrate the service.

In the following, we deal only with the later direction. Of course people who think about very specific services, that is to say services which fit only to one or to a very small number of applications should integrate them using classical AOP techniques. But as far as it concerns services that should be highly adaptable and which address a wide range of applications, then we are convinced that it is better to develop an approach based on the insertion of hyper-generic treatment which are orthogonal to the language to be extended (with the service).

For example, let us add to the *Java* language, a set of capabilities for handling persistent objects that may be adapted to different contexts, from very basic to complex management of persistent objects; then it may worth to define an *OFL* service which relies on a set of specific parameters allowing to adapt exactly the behaviour of the language extension to the needs of the application. For example, it may be interesting to handle differently applications whether they use large collections of simple objects or small collections of complex objects.

4.1 An intuitive approach to create a new service

When building a new service several steps are required:

- to think about the scope of the service and its limitations. So that we know if it worths to define an OFL-service instead of using classical AOP.
- To identify the set of parameters that will parametrise the service to be added. Building a new service for handling persistence could require the definition of parameters for handling object loading and transaction management policies. This step requires to have a good knowledge of the state of the art related to the service objectives.
- To identify properties that should be included to the reification of description and to the reification of instances of these descriptions. For handling persistence, an instance of description will get a `persistent object identifier` and a `version number` will be associated to a description.
- According to the parameters that had been found, to check if there is any need to build some specific kinds of description or some specific kinds of relationship. A service which implement basic persistence handling does not require specific components but a service dealing with the handling of object evolution will certainly lead to the definition of a new kind of description (`version`) and to one or several kinds of relationship (`is-a-version-of`, etc.).
- To identify possible specific atoms, that is to say not customisable entities included into the model but whose semantics may be captured only by new assertions. For example, handling persistence may require a new kind of atom which associates an object persistent identifier (POI) to its address in volatile memory (VOI). Some assertions should be defined in order to catch how to synchronise the persistent object with its volatile copy.
- To write pieces of code that implement the operational semantics of those parameters. They should be integrated to the set of actions that are specified by the *OFL* environment (at run -time or statically) in order to handle application according to the characteristics, parameters and properties which had been added (see section 4.2 for some more details).

4.2 Extension of OFL-core

As it is shown in figure 2[2] the modelling of a new service strongly depends on the architecture of the *OFL* model. We provide hereafter some explanation about the extension of OFL-core with one new service. To extend the model with several services is not a problem since all services are fully independent (see the gold rules of our extension below).

[2]All arrows of the first figure haven't been duplicated here and `an-application` is the same in the two figures.

OFL-aspects For each OFL-atom of OFL-core we get one aspect which contains its modification, that is to say new characteristics. It is also possible to define new atoms which are specific to the service.

For each OFL-concept of OFL-core, we get one aspect which contains its modification, that is to say:

- New parameters and characteristics related to the concept which are specific to the service to be added.
- New assertions that have to be added to the concept in order to take into account new parameters and properties as well as some new constraints about new or modified atoms.
- New methods that contain semantics that have to be included in actions defined in the concept. It is not possible to add any action but only to insert additional code at specific location of the existing ones. Moreover, the added semantics may be influenced by values of parameters or by characteristics whatever they come from OFL-core or from OFL-aspects, but they may only modify characteristics of OFL-aspects. This ensures the orthogonality of the service according to the OFL-core model.

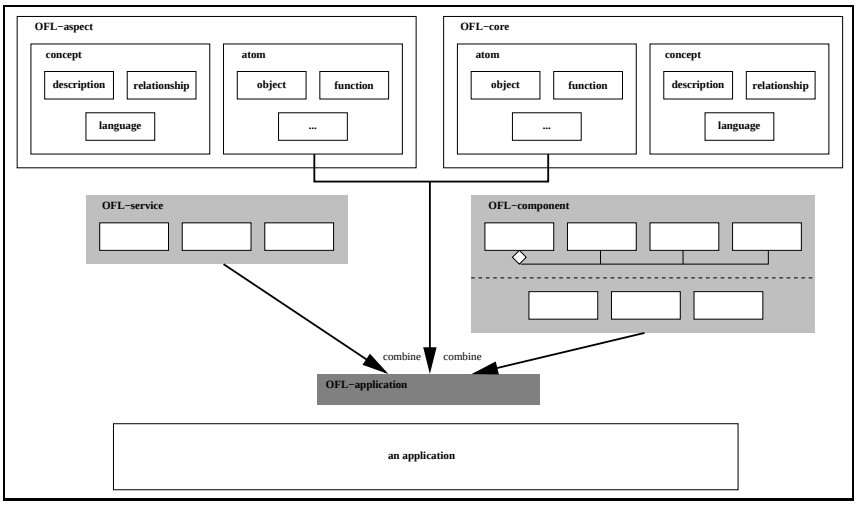

Figure 2: Extension of OFL-core with one service

OFL-services The choice of the values of parameters associated to OFL-aspects correspond to the customisation of the service which implement the particular needs of the application. We recall that it is mandatory to have a deep study of the application domain needs in order to provide a consistent set of parameter values. OFL-services represent instances of OFL-aspects (OFL-services may differ one from the others only according to parameter values). With such an approach we could get only one service associated to one concern of application, for example one service of persistence, its set of parameter allowing to adapt the service to the need of application.

Specific OFL-components The implementation of some new services such as evolution require also the description of new OFL-component, for instance the management of object evolution needs at least a new import relationship like “is a version of”. Some pieces of the semantics added to OFL-core will handle these specific components explicitly, so that these components are strongly related to the service[3].

OFL-application Its aim is to put OFL-services and OFL-components together. An application is described using the OFL-components related to one language. As it is shown in figure 2, an application may needs to use one service so that it has to extend the OFL-components initially associated to the language with the OFL-services which correspond to the service implementation. The OFL-application is the OFL-atom which provides a way to put together: the OFL-services corresponding to the customisation of the

[3]Structure and behaviour of specific OFL-components are not different from any other OFL-component.

service which match the application requirements, the OFL-components especially built for this service and the atoms which refine the reification of the entities describing the application[4].

Our gold rules The protocol used to modify actions which are part of OFL-core concepts is based on AOP techniques and ensures that following rules are satisfied:

1. Any service may be added (or removed) without any consequences for other services.
2. Any service may be added (or removed) on any order.
3. Service can't reference another one.
4. Services can reference parameters and characteristics of OFL-core.
5. Services can't modify characteristics of OFL-core.
6. A service is integrated in such a way that it is transparent for an existing application.
7. Each program can *choose* to apply or not apply each service.

5 Conclusion and perspectives

To build reusable services, whose definition is orthogonal to the application, is an exiting issue. We are convinced that the separation of concerns and especially the AOP is an interesting approach to achieve it as far as it deals with specific services dedicated to one application or to a small set of applications. But we are also convinced that general orthogonal services, that has to be adapted to the use of a large range of application should be integrated at the language level. We suggest that the integration is made through customisable entities that allow to modify the semantics of the language and as consequence, the behaviour of the application. We are interested in the near future by implementing this approach using existing AOP environment such as AspectJ [KHH+01].

References

[CCCL01] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Hyper-généricité pour les langages à objets : le modèle OFL. In *Conférence LMO 2001 (Langages et Modèles à objets)*. Hermes Science Publications, janvier 2001.

[CCL01] A. Capouillez, P. Crescenzo, and P. Lahire. OFL : l'hyper-généricité au service du méta-programmeur (Application à Java). Technical report, Laboratory *Informatique, Signaux et Systèmes de Sophia Antipolis*, mars 2001.

[Fla99] D. Flanagan. *Java in a Nutshell: a Desktop Quick Reference*. O'Reilly, 3rd edition, December 1999.

[GJSG00] J. Gosling, B Joy, G Steele, and Bracha G. *The Java Language Specification*. The Sun Microsystems Press Java Series. Sun Microsystems, June 2000.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *the European Conference on Object-Oriented Programming (ECOOP)*, June 2001.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2nd edition, 1997.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., 3rd edition, 1997.

[4]We make no assumption about possible additional features for building a service according to one or several other services already implemented.