Towards a Reference Model for Implementing the Fractal Specifications for Java and the .NET Platform

Lionel Seinturier$^{(1)}$, Nicolas Pessemier$^{(1)}$, Clément Escoffier$^{(2)}$, Didier Donsez$^{(2)}$
$^{(1)}$ INRIA Futurs - LIFL, Project Jacquard/GOAL, 59655 Villeneuve d'Ascq, France
$^{(2)}$ Univ. Grenoble, LSR-IMAG, Adele Team, 38041 Grenoble, France
{seinturi,pessemie}@lifl.fr, {Clement.Escoffier,Didier.Donsez}@imag.fr

1 Introduction

So far, several implementations of the Fractal specifications have been proposed. These implementations propose frameworks for programming with Fractal in a target language (Java, C, Smalltalk, C++). The general principles of implementing the specifications are common to all these frameworks. However, as far as we know, no concrete piece of software or no common set of internal interfaces have ever been shared between them.

In this paper, we report on a reference model which has been set up to support several implementations on the Fractal Specifications. This model has been derived to build a Java personality called AOKell, and FractNet, a personality for the languages of the .NET framework. The originality of this model is to be based on the concepts of Aspect-Oriented Programming (AOP) [1].

Section 2 presents the general architecture of the reference architecture. Section 3 reports on the two personalities, AOKell and FractNet. Section 4 shows some performance measurements. Section 5 concludes this paper.

2 Architecture

2.1 Background

Two dimensions can generally be found in a Fractal implementation: the business dimension and the control dimension. The former is responsible for implementing the core functionalities of the application, while the latter provides a level of supervision and management on the application.

Fractal defines some artefacts to implement the business dimension with components: the notion of a provided interface, a required interface, a component type or an interface type. The control dimension, so called controllers, deals with functionalities such as starting/stopping the components, managing their bindings, their attributes, or the containment hierarchy. The scope of the control dimension is fully open: there is no restriction on what the control dimension should or shouldn't do. It is up to the framework developer to design and implement the control functions s/he needs.

The role of a Fractal framework implementor is then: (1) to support the API defined in the specifications, (2) to provide some structures for implementing controllers, (3) to provide some mechanisms for integrating the control and the business dimensions, i.e. to apply the control functions to the components. For example, Julia [2], the Fractal reference implementation, uses mixins [3] to perform this integration. The originality of the approach presented in this paper is to use aspect-oriented programming (AOP) [1] for integrating the control dimension with the business one.

Aspects [1] are software entities modularizing concerns which are said to be crosscutting. The code of such concerns is not properly modularized in one unique entity (class, procedure, function, etc.), but is spread around many different locations. This phenomenon, known as code scattering, hinders the development and the maintenance of applications. AOP promotes a way of remodularizing these concerns, with the notion of an advice code, and a way of integrating them in the rest of the application, with the notion of a pointcut. By this way, AOP complements existing programming styles, object-oriented, procedural, functional, etc., by removing scattering and producing software which is more modular. AOP has been the subject of many studies in the past years and as been applied to many domains, including middleware (just to name a few works in this area, see for example [4], [5] or [6]).

2.2 Aspect-Oriented Architecture

Our reference model defines three layers: the two above-mentioned dimensions, application and control, and a middle layer which is aspect-based (see Figure 1). This aspect layer is responsible for modularizing and integrating the control functions into the application. This layer supervises and controls the applications and delegates the concrete realizations of these functions to the control layer.

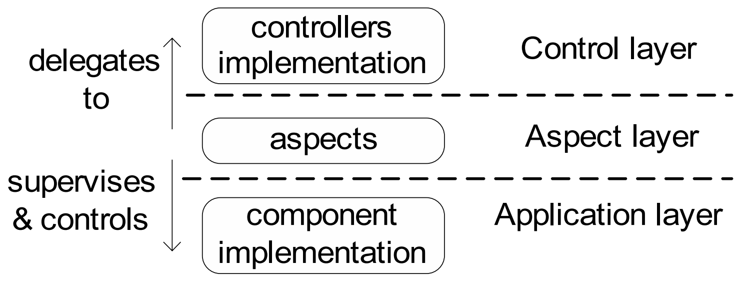

Figure 1: High-level view of the reference model.

The aspect layer defines one aspect per control functions. For example, the seven basic controllers of Fractal (attribute, binding, factory, component, content, naming, super) are each associated to an aspect. Any new controller which needs to be added must comes with its aspect. The aspects perform two basic tasks: (1) code advising, (2) code injection. Advising is an AOP related term which refers to the ability of intercepting some events of the application (such as method calls, method executions, field writes, etc.) and performing treatments before and/or after these events. Code injection refers to the ability of extending the implementation of a component with new methods, interfaces or fields.

Code advising is used in the aspect layer to control and supervise the application layer. For example, method executions can be intercepted, blocked or released depending on a particular life cycle policy. Code injection is used to enhance the functionalities of a component with new interfaces, such as an interface for managing binding, names, or any other control interface.

In our model, while aspects supervise and control, they do not realize themselves the control functions. They delegate it to the control layer. For example, the aspect in charge of the management of bindings, injects a stub for the `BindingController` interface which delegates the implementation of this interface to an object of the control layer. This pattern is a frequently used practise of AOP: it allows decoupling the integration logic (aspect layer) from the concrete realization of this logic (control layer).

This architecture are been declined into two implementations, one for Java (AOKell) and one for the .NET platform (FractNet). They are presented in the next sections. Furthermore, some parts of AOKell have been reused in FractNet. The performances of these implementations are summarized in Section 4.

3 Implementations

This section reports on the two implementations, AOKell and FractNet, which have been derived for the previous reference model. They are briefly presented in sections 3.1 and 3.2. Section 3.3 provides a short description of the code elements which have been shared between the two implementations.

3.1 AOKell

AOKell[1] is the Java implementation of our reference model. AOKell performs a compile-time (CT) integration of control functionalities into components. As a matter of comparison, Julia [2], the Fractal reference implementation, performs a load-time (LT) integration based on bytecode engineering performed with the ASM library [7]. While LT integration is somewhat more flexible than CT integration, CT integration leads to a solution which is more type safe and easier to debug.

Two versions of the aspect layers are available with AOKell: one based on AspectJ and one based on Spoon.
AspectJ [8] is the leading tool for AOP. A compiler is available and many tools, including plugins for IDEs, have been developed. While the most common usage of AspectJ is for weaving aspects at compile-time, AspectJ can also be used as a load-time weaver.
Spoon [9] is a general purpose tool for transforming Java programs. A strongly-type template mechanism is available for writing any kind of transformations including the ones (code advising and code injection) which are similar to the ones performed by aspect weavers. Spoon is architectured as a back-end of the javac compiler.
The functionalities of the two versions are strictly equivalent. However, as we will show it in Section 4, the Spoon-based version of AOKell performs better than the AspectJ-based one. Indeed, being a general purpose tool for code transformation, Spoon allows generating code which is more aggressively optimized.

As an additional feature, AOKell allows developing the control layers as assemblies of component. For that, we introduce the notion of a control component. A control component implements a given control function (e.g. managing bindings, lifecycle, attributes, etc.) and is integrated into the application with an aspect. The idea is that controllers are seldom autonomous, but that, most of the time, they require the collaboration of other control components. When programming these controllers as plain old Java objects (POJO), the interaction schemes between controllers end up being dug into the code as references to other objects. This hinders modularity and leads to code which is poorly structured. The idea is then to apply to the control layer the same principles which have been applied to applications: capture the architecture of the control layers with an ADL (in our case Fractal-ADL), and implement control membranes as assemblies of control components. AOKell provides a set of 13 componentized control membranes, corresponding to the ones which can be found in Julia. The development of a new control membrane is then a matter of writing a new assembly of control components. Although the componentization of membranes is available with AOKell, this is not a mandatory feature: developers can still write controllers and membranes as POJOs.

3.2 FractNet

FractNet[2] is the implementation of our reference model for the .NET platform. While the aspect layer is specific to FractNet, the control layer of AOKell is reused as this with FractNet. The Java implementation of this layer is compiled with the Visual J# compiler to produce a regular .NET assembly which can be linked with the rest of the application.

The aspect layer of FractNet is implemented with the AspectDNG [10] aspect weaver. AspectDNG performs a compile-time weaving of aspects on compiled .NET assemblies. This ap-

[1]`http://fractal.objectweb.org`
[2]`http://www-adele.imag.fr/fractnet`

proach has two main advantages: an application can be woven even if its source code is not available, and AspectDNG can weave applications written in any language supported by .NET (C#, J#, VB.NET, Managed C++, etc.). By this way, Fractal applications with FractNet can be written in any languages that can be compiled into a .NET assembly.

3.3 Shared code elements

The source code of AOKell and FractNet is organized in three main packages (see Figure 2): **glue** implements the aspect layer defined in the reference architecture, **lib** implements the control layer, **component** is a library of predefined components (this library contains the bootstrap component). A fourth package, **tools**, provides some utility tools which are not mandatory for building and running a Fractal application with AOKell or FractNet.

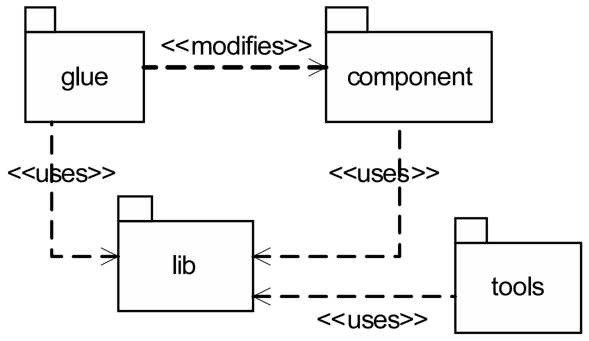

Figure 2: AOKell and FractNet packages organization.

As summarized in Table 1, the glue package is specific to AOKell and FractNet, whereas the component and lib packages are common to both implementations. These packages are written in Java. They are compiled with the standard javac compiler in the case of AOKell, and with the Visual J# compiler in the case of FractNet.

	AOKell	FractNet	% (lines of source code)
glue	AspectJ or Spoon	AspectDNG	12%
component	Java		1%
lib			82%
tools			5%

Table 1: AOKell and FractNet code sharing.

The figures given in the last column of Table 1 correspond to the percentages in terms of lines of source code for each package. AOKell with Spoon contains about 10,000 lines of code. The glue package, which contains 12% of the source code in this case, is the only one to be specific to either AOKell or FractNet. The remaining 88% are shared between the two implementations.

4 Performance Measurements

This section reports on a microbenchmark which has been conducted with a simple application containing two components: a client component and a server component. The server component provides an interface with eight methods. Each method owns a different signature, either without parameters, or with primitive parameters, or with object references parameters, and/or with return types.

The measures are taken on 2Ghz Pentium 4 PC running Windows XP Pro, Sun JDK 1.5.0 and the Microsoft .NET Framework 2.0. A warm-up phase is performed before taking measures to avoid bootstrapping and class loading costs. The test consists of series of calls emitted from the client component to the server component. In table 2, the figures correspond to the times taken by 8,000,000 calls (1,000,000 per method defined in the interface provided by the server component). The given figures correspond to the average value of 4 runs.

	Operation execution time	
	no lifecycle	lifecycle
Fractal/FractNet	228ms	264ms
Fractal/AOKell 2.0 (Spoon based)	212ms	221ms
Fractal/Julia 2.1.1 (option MERGEALL)	234ms	396ms
Fractal/AOKell 1.0 (AspectJ based)	212ms	443ms

	Operation execution time	
	without interception	with interception
Pure .NET 2.0	189ms	
Pure Java 1.5.0	122ms	172ms
AspectJ 1.2.1		206ms
JBoss AOP 1.1.1		1046ms

Table 2: Cost of invoking and executing an operation (x 8,000,000).

This microbenchmark provides two series of measures depending on whether the lifecycle controller is activated or not. As a matter of comparison, we also provide some reference costs in pure Java or pure C#, and with three interception techniques: AspectJ, JBoss AOP and statically generated proxy classes in Java.

The conclusion which can be drawn from this microbenchmark is that, in the case of AOKell, Spoon delivers better performance than AspectJ. The difference between AOKell and Julia mainly comes from some difference in the semantics of the lifecycle controller implemented by these two frameworks. The performances of the Java and .NET implementations of Fractal are roughly similar, the Java ones being a little more performant.

5 Conclusion

This paper presents two implementations of the Fractal specifications: one, AOKell, for programming Fractal applications with Java, and one, FractNet, for programming with the languages of the .NET framework. Compared to other existing implementations of Fractal, the novelty of our approach is to provide a reference model which defines some general design principles for deriving frameworks implementing the specifications. The originality of this model is to be based on concepts of aspect-oriented programming.

Three layers can be found in the model: the application layer, the aspect layer and the control layer. The implementation of the control layer is shared between the two implementations, leading to a high level of code reused: about 88% of the FractNet code comes from AOKell (written in Java, this code is compiled for .NET with the Visual J# compiler). The aspect layer is specific to each implementation. Two versions of the layer exist for AOKell, one written with AspectJ [8], and another one written with Spoon [9] (the latter performs better than the former). The aspect layer of FractNet is written with AspectDNG [10]. The benefit of using AspectDNG is that any .NET assembly can be woven with the control layer. Thus, programming Fractal applications with FractNet can be conducted, not just in one language, but in any language which compiles code into .NET assemblies.

As a matter of future work, we plan to use our frameworks, both AOKell and FractNet, to build dynamic service-oriented platforms [11]. Another direction would consist in defining a SPI (Service Provider Interface) associated to the reference model. This SPI would provide a more concrete framework for reusing finer grained piece of code between different Fractal implementations. Concerning FractNet, we plan to migrate to another .NET aspect weaver since the AspectDNG project seems to be discontinued.

Acknowledgments

The work on AOKell is partially funded by France Telecom under the external research contract #46131097.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE-7)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, May 2004.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA'90)*, volume 25 of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.

[4] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 56–65. ACM Press, 2004.

[5] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 130–139. ACM Press, 2003.

[6] U. Kulesza and D. Silva. Reengineering of the JaWS web server design using aspect-oriented programming. In *Workshop on Aspects and Dimensions of Concerns at ECOOP'00*, July 2000.

[7] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Journées Composants 2002 (JC'02)*, November 2002. `asm.objectweb.org/current/asm-eng.pdf`.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[9] R. Pawlak. Spoon: Annotation-driven program transformation the AOP case. 1st Workshop on Aspect-Orientation for Middleware Development @ Middleware'05, November 2005. `spoon.gforge.inria.fr`.

[10] T. Gil and J.-B. Evain. *AspectDNG*. DotNetGuru, 2005. `www.dotnetguru.biz/aspectdng/`.

[11] C. Escoffier, D. Donsez, and R. S. Hall. Developing an osgi-like service platform for .NET. In *3rd IEEE Consumer Communications and Networking Conference (CCNC'06)*. IEEE, January 2006. `www.ieee-ccnc.org/2006/`.