Using Attribute-Oriented Programming to Leverage Fractal-Based Developments

Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, Philippe Merle

▶ **To cite this version:**

Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, Philippe Merle. Using Attribute-Oriented Programming to Leverage Fractal-Based Developments. 5th International ECOOP Workshop on Fractal Component Model (Fractal'06), Jul 2006, Nantes, France. hal-01353552

HAL Id: hal-01353552

https://inria.hal.science/hal-01353552

Submitted on 11 Aug 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 Attribute-Oriented Programming to Leverage Fractal-Based Developments

Romain Rouvoy Nicolas Pessemier Renaud Pawlak Philippe Merle
JACQUARD Project - INRIA Futurs
LIFL - University of Lille I
59655 Villeneuve d'Ascq Cedex, France
{romain.rouvoy, nicolas.pessemier, renaud.pawlak, philippe.merle}@inria.fr

Abstract

This paper presents the *Fraclet* annotation framework. The goal of Fraclet is to leverage Component-Based Software Engineering based on the Fractal component model using *Attribute-Oriented Programming*. We show that, using Fraclet, about 50% of the hand-written program code can be saved without loosing the semantics of the application, while the rest of the program code is automatically generated.

1. Introduction

Component-Based Software Engineering (CBSE) is concerned with the development of highly reusable business components which declare contractually specified interfaces to communicate with each other. CBSE facilitates the development of high quality applications with shorter development cycles and reduces coding effort. However, in practice, it appears that the component developer's task is not only devoted to the design and implementation of the business logic of the applications but also to the integration of redundant and error prone technical properties.

A convenient way to address this issue is to use *Attribute-Oriented Programming* (@OP) techniques. @OP proposes to mark program code with metadata to clearly separate the business logic from a domain-specific logic (typically technical properties). @OP is gaining popularity with the recent introduction of annotations in Java 2 standard edition (J2SE) 5.0 [7] or in XDoclet [17], and attributes in C# [5]. Recently, the Enterprise Java Bean (EJB) 3.0 specification extensively uses annotations to make EJB programming easier [4]. The Service Component Architecture (SCA) component implementation model provides a series of annotations which can be placed in the code to mark significant elements of the implementation which are used by the SCA runtime [8].

The contribution of this paper is to present *Fraclet*, an annotation-based framework using @OP to leverage the Fractal component developer's task. Fraclet provides a set of dedicated annotations to mark the Fractal-related technical properties in the program code. To achieve this goal, Fraclet introduces a seamless design process for Fractal component developers. Thanks to @OP, most of the component artifacts are automatically generated.

[copyright notice will appear here]

Fraclet is developed to deal with the Java implementations of the Fractal component model. We have experimented two different, but functionally equivalent, implementations of the Fraclet annotation framework. The first one uses XDoclet and Velocity to define the code generators and to produce the various artifacts required by the Fractal component model. The second one uses Spoon [12], a Java 5-compatible processing tool, which supports the processing of Java 5 annotations. The Fraclet developer can then take advantage of Java 5 type safety and annotations. Regardless of the implementations, we show that, using Fraclet, about 50% of the hand-written program code can be saved without loosing the semantics of the application, while the rest of the program code is automatically generated and continuously integrated.

The remainder of this paper is organized as follows. Section 2 introduces the Fractal component model and the concepts related to Attribute-Oriented Programming. Section 3 introduces the motivation of our work and illustrates it on a simple example. Section 4 presents the Fraclet annotation framework. Section 5 addresses the implementation issues related to our work. Section 6 performs some evaluations. Section 7 compares with related work. Finally, Section 8 concludes and presents some perspectives.

2. Background

This section first introduces the Fractal component model and then it presents the Attribute-Oriented Programming principles.

2.1 The Fractal Component Model

The hierarchical Fractal component model uses the usual *component*, *interface*, and *binding* concepts [2]. A component is a runtime entity that conforms to the Fractal model. An interface is an interaction point expressing the provided or required methods of the component. A binding is a communication channel established between component interfaces. Furthermore, Fractal supports *recursion with sharing* and *reflective control* [3]. The recursion with sharing property means that a component can be composed of several sub-components at any level, and a component can be a sub-component of several components. The reflective control property means that an architecture built with Fractal is reified at runtime and can be dynamically introspected and managed. Fractal provides an Architecture Description Language (ADL) [11], named *Fractal ADL*, to describe and automatically deploy component-based configurations.

The Fractal component model is already applied at any software granularity from operating systems (*e.g.*, Think [6]) to middleware (*e.g.*, GoTM [13], DREAM [10]), and application servers (*e.g.*, JOnAS à la carte [1]).

Figure 1 illustrates the different entities of a typical Fractal component architecture. Thick black boxes denote the controller part of a component, while the interior of the boxes corresponds to the content part of a component. Arrows correspond to bindings, and tau-like structures protruding from black boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. External interfaces appearing at the top of a component represent reflective control interfaces such as the component controller (c), the lifecycle controller (lc), the binding controller (bc), the content controller (cc), and the attribute controller (ac) interfaces.

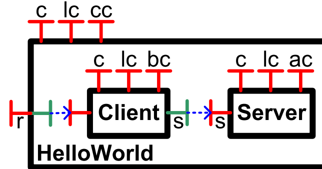

Figure 1. The Fractal component HelloWorld.

The simple application depicted in Figure 1 will be used in this paper to illustrate the overhead induced by CBSE. This application is composed of a component Client and a component Server. These two primitive components are contractually bound by the interface Service, named s, that describes the required and the provided operations of the components Client and Server, respectively. The components are included in the composite component HelloWorld, which exports the interface Runnable, named r, provided by the component Client.

2.2 Attribute-Oriented Programming

Attribute-oriented programming is a program-level marking technique. Basically, developers can mark program elements (*e.g.*, classes, methods and fields) with *annotations* to indicate that they maintain application-specific or domain-specific semantics [16]. For example, some developers may define a `logging` annotation and associate it with a method to indicate the method should implement a logging function, while other developers may define a `web service` annotation and associate it with a class to indicate the class that would implement a Web Service. Annotations separate application's business logic from middleware-specific or domain-specific semantics (*e.g.*, logging and web service functions).

By hiding the implementation details of those semantics from program code, annotations increase the level of programming abstraction and reduce programming complexity, resulting in simpler and more readable programs. The program elements associated with annotations are transformed to more detailed programs by a supporting tool (*e.g.*, generation engine). For example, a generation engine may insert a logging program into the methods associated with a `logging` annotation. The dependencies towards the underlying middleware are replaced by annotations, acting as weak references. This means that any evolution of the underlying middleware is taken into account by the generation engine and let the program code unchanged.

@OP lets you apply *continuous integration* in CBSE. Continuous integration allows the developer to generate the middleware artifacts at any step of the development of the component. Developers concentrate their editing work on only one source file per component. The deployment metadata is continuously integrated without worrying about updating it. When the development of a component consists of several files, @OP allows the developer to maintain only one of them and the other files are generated automatically. Besides, working with only one file per component gives a better overview of the program code to the developer. Therefore, the developer can concentrate on the business logic and reduce the development time drastically.

3. Motivation

Although CBSE provides more modularity, configurability, and reusability properties to applications, the use of a given component model introduces also more complexity, more verbosity and redundancy in the information expressed by the developer. Indeed, developing an application using CBSE principles requires to take into account several concerns. These concerns, which are not always related to the business of the application, are illustrated in this paper with the application HelloWorld depicted in Figure 1.

When developing such an application, the first step consists in describing the operations of the interface `Service`. Figure 2 introduces the operation `print` that takes a message as parameter (Line 2). This contract ensures that the components Client and Server can communicate.

```
1 public interface Service {
2   void print(String message);
3 }
```

Figure 2. Implementing the interface Service.

In the second step, the class `Client` (see Figure 3) implements the interfaces `Runnable` and `BindingController` (Lines 4–5). The method of the interface `Runnable` concerns the business code of the application (Lines 8–10), and the methods of the interface `BindingController` deals with the technical code related to management of client interface references (Lines 12–24).

```
4 public class Client
5   implements Runnable, BindingController {
6   private Service s; // Interface Service named s.

8   public void run() {
9     this.s.print("hello world");
10  }

12  public String[] listFc() {
13    return new String[] { "s" };
14  }
15  public Object lookupFc(String cItf) {
16    if (cItf.equals("s")) return this.s;
17    return null;
18  }
19  public void bindFc(String cItf, Object sItf) {
20    if (cItf.equals("s")) this.s = (Service) sItf;
21  }
22  public void unbindFc(String cItf) {
23    if (cItf.equals("s")) this.s = null;
24  }
25 }
```

Figure 3. Implementing the component Client.

In the third step, the class `Server` (see Figure 4) implements the interfaces `Service` and `ServerAttributeController` (Lines 34–35). The method of the interface `Service` represents the provided operation (Lines 39–42). The methods of the interface `ServerAttributeController` are dedicated to the configuration of the component attributes (Lines 44-55). Besides, the interface `ServerAttributeController` should define the methods for handling the attributes `header` and `count` defined by the component Server (Lines 26–32).

Once the components Client and Server are implemented, their composition can be described using Fractal ADL (see Figure 5). The definitions `Client` and `Server` describes the components Client and Server, respectively (Lines 1–17). The definition `HelloWorld` contains instances of a component Client and a

```java
26 public interface ServerAttributeController
27         extends AttributeController {
28   String getHeader();
29   void setHeader(String header);
30   int getCount();
31   void setCount(int count);
32 }

34 public class Server
35   implements Service, ServerAttributeController {
36   private String header; // Attribute header.
37   private int count;     // Attribute count.

39   public void print(String msg) {
40     for (int i = 0; i < this.count ; ++i)
41       System.out.println(this.header + msg);
42   }

44   public String getHeader () {
45     return this.header;
46   }
47   public void setHeader (String header) {
48     this.header = header;
49   }
50   public int getCount() {
51     return this.count;
52   }
53   public void setCount(int count) {
54     this.count = count;
55   }
56 }
```

Figure 4. Implementing the component Server.

component Server configured with their attribute values (Lines 22–25). The interface `r` of the component Client is exported and the `client` and the `server` are bound together through the interface `s` (Lines 26–27).

```xml
1 <definition name="Client">
2   <interface name="r" role="server"
3              signature="Runnable"/>
4   <interface name="s" role="client"
5              signature="Service"/>
6   <content class="Client"/>
7 </definition>

9 <definition name="Server" arguments="header,count">
10   <interface name="s" role="server"
11              signature="Service"/>
12   <content class="Server"/>
13   <attributes signature="ServerAttributeController">
14     <attribute name="header" value="${header}"/>
15     <attribute name="count" value="${count}"/>
16   </attributes>
17 </definition>

19 <definition name="HelloWorld">
20   <interface name="r" role="server"
21              signature="Runnable"/>
22   <component name="client"
23              definition="Client"/>
24   <component name="server"
25              definition="Server('->',1)"/>
26   <binding client="this.r" server="client.r"/>
27   <binding client="client.s" server="server.s"/>
28 </definition>
```

Figure 5. Describing the components composing HelloWorld.

Figures 2 to 4 represent the minimal piece of Java code required to implement the application HelloWorld using the Fractal component model. Nevertheless, when considering the 56 lines of Java code of this example, 13 lines are related to the control of client interfaces (`BindingController` interface implementation) and 19 lines are related to the control of the component attributes (description and implementation of the interface `ServerAttributeController`). This means that 57% of the Java code handles the technical concerns imposed by the Fractal component model.

Figure 5 represents the minimal piece of XML definitions required to describe the application HelloWorld using Fractal ADL. This language targets to simplify the composition of Fractal components by the use of an higher-level language dedicated to the definition of architectures. Nevertheless, 17 lines of the 28 lines composing the application describe entities that are still expressed in Java (the primitive components Client and Server). This means that 61% of the descriptors are redundant with the information already present in the component implementations.

In summary, the global overhead of the component model is evaluated in this example to 58%. This number encloses either the technical code specific to Fractal, and the information duplicated between the ADL and the implementation language. Concerning the evolution of the application, the modification of the component content (*e.g.*, adding an attribute or a binding) is a time consuming and repetitive task, which can be subject to errors.

4. The Fraclet Annotation Framework

Given that the main objective is to leverage the Fractal development process, we choose to apply @OP to drastically reduce the size of the program code by factorizing the redundant information as well as the technical properties imposed by the Fractal component model. Therefore, Fraclet provides various annotations to clarify the specificities of the Fractal programming model and the associated generators to produce the Fractal artifacts. After presenting the objectives, an overview of the Fraclet annotation framework is provided.

4.1 Objectives of Fraclet

Fraclet should be designed to support program code evolution. This means that, when considering an incremental development process, Fraclet should be able to produce up to date artifacts at any development step.

Then, Fraclet should not introduce an overhead to the program execution compared to an handwritten program code. This means that the efficiency of the execution support should not be affected by the use of Fraclet.

The design of Fraclet is driven by a concern to facilitate its integration by developers that are already using Fractal. This means that, Fraclet should reuse the concepts introduced in the Fractal component model and the associated tools (*e.g.*, Fractal ADL).

Finally, Fraclet should maximize the size of the generated program code, while minimizing the use of annotations. This means that Fraclet should deduce as much as possible of the relevant information from the program code to produce handwritten-equivalent artifacts.

4.2 Overview of the Framework

The Fraclet annotation framework is composed of two parts: The annotations and the generators. An annotation is associated to a given program element, while a generator uses one or several annotated program codes to produce an artifact.

Annotations. Fraclet reifies the Fractal technical concepts as a set of dedicated annotations. Basically, an annotation is composed of an identifier and a set of parameters. The value of a parameter can be required, statically defined among a list of allowed values, or deduced from the program code. Table 1 describes the four annotations defined in Fraclet. The @Interface annotation enhances the definition of a Java interface with a Fractal identifier. The @Binding, @Attribute, and @Controller annotations add to a Java

Annotation	Location	Parameter	Description	Contingency	Default Value
@Interface	Class	*name*	server interface name	required	-
		signature	server interface signature	optional	signature of the Java interface
@Binding	Field	*name*	client interface name	optional	name of the Java field
		signature	client interface signature	optional	signature of the Java field
		cardinality	client interface cardinality (`singleton`|`collection`)	optional	`singleton`
		contingency	client interface contingency (`mandatory`|`optional`)	optional	`mandatory`
@Attribute	Field	*name*	component attribute name	optional	name of the associated field
		value	component attribute initial value	optional	-
		argument	argument name in the component definition	optional	attribute name
@Controller	Field	*name*	component controller name to access	optional	`component`

Table 1. Overview of Fraclet annotations

field the semantics of a client interface, a component attribute, and a component controller, respectively.

Generators. Once the program code is annotated, the Fractal artifacts are produced by four generators. Figure 6 presents the architecture of the generation engine used to produce the various component artifacts. This generation engine reifies the annotated program code as a model in memory. The generators use the program code model to produce either Java program code or XML definitions. Then, the handwritten and the generated program code are compiled by the Javac tool. Finally, the compiled code and the application descriptors are used by the Fractal ADL tool to deploy the application.

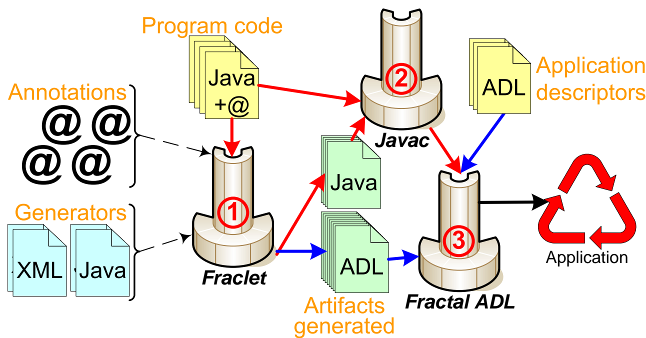

Figure 6. Overview of Fraclet generation engine.

Fraclet provides two Java program code generators and two XML definition generators:

Attribute controller. It is a Java generator that produces, for each Fractal component defining an attribute, an interface that handles the attributes defined for the Fractal component. This interface contains a getter and a setter method for each attribute defined for the Fractal component.

Component glue. It is a Java generator that produces, for each Fractal primitive component, the non-functional code. This encloses the program code required to implement the methods of the attribute controller interface generated previously and the methods defined in the binding controller interface.

Primitive definition. It is an XML generator that produces, for each Fractal primitive component, a definition of the primitive component compliant with the Fractal ADL tool. This definition defines the interfaces provided and required by the component, the attributes with their initial value or their argument, and the name of the content class implementing the component.

Abstract composite definition. It is an XML generator that produces, for each Fractal primitive component with at least a client interface, a definition of a composite component containing the associated primitive component, and abstract definitions of its dependencies. The bindings between the client interfaces of the component and the server interfaces of the dependencies are automatically declared. The server interfaces of the component are automatically exported as server interfaces of the composite component.

4.3 Revisiting HelloWorld with Fraclet

In this section, the application HelloWorld introduced in Section 2 is revisited using Fraclet.

In Figure 7, the interface `Service` is annotated with the @Interface annotation to define its default Fractal name (Line 1).

```
1 /** @Interface name="s" */
2 public interface Service {
3   void print(String message);
4 }
```

Figure 7. Annotating the interface Service.

The content class `Client` is annotated with the annotation @Interface to specify that the component provides an interface `Runnable` whose Fractal name is `r` (Line 5 in Figure 8). The annotation @Interface can mark a content class if the interfaces implemented by this class are not already marked (*e.g.*, an interface defined in an external library). Then, the `service` field is marked with the annotation @Binding and named `s` (Line 7). The signature of the resulting client interface is deduced from the signature of the field, which is `Service` in this case.

```
5 /** @Interface name="r" signature="Runnable" */
6 public class Client implements Runnable {
7   /** @Binding name="s" */
8   protected Service service;

10  public void run() {
11    this.service.print("hello world");
12  }
13 }
```

Figure 8. Annotating the component Client.

The content class `Server` automatically inherits from the `Service` annotations (see Figure 9). The fields `header` and `count` are marked as Fractal attributes with the annotation @Attribute (Lines 15–18). The name and the argument of these attributes depends on the name of the marked field.

Therefore, the program code related to the management of component attributes and bindings is no more tangled with the business code of the component.

Figure 10 defines the composition of the application HelloWorld. Given that the abstract composite generators defines an abstract definition of the application, the developer writes the definition `HelloWorld` by extending the abstract composite definition

```
14 public class Server implements Service {
15   /** @Attribute */
16   protected String header;
17   /** @Attribute */
18   protected int count;

20   public void print(final String msg) {
21     for (int i = 0; i < this.count ; ++i)
22       System.out.println(this.header + msg);
23   }
24 }
```

Figure 9. Annotating the component Server.

`ClientComp` (Line 1) and specifying the component `Server` definition named s (Line 2).

```
1 <definition name="HelloWorld" extends="ClientComp">
2   <component name="s" definition="Server('->',1)"/>
3 </definition>
```

Figure 10. Describing the component HelloWorld.

Figures 7 to 10 show that our approach reduces drastically the size of the handwritten program code, while preserving the semantics of the application HelloWorld. An empirical evaluation of our approach is presented in details in Section 6.

5. Implementation Issues

This section presents two functionally equivalent implementations of the Fraclet annotation framework. The first defines XDoc annotations and uses the XDoclet generation engine to produce the various artifacts required by the Fractal component model. The second defines Java 5 annotations and uses the Spoon transformation tool to enhance the handwritten program code with the Fractal technical properties.

5.1 Implementing Fraclet with XDoclet

XDoclet overview. XDoclet[1] is an open program code generation engine. It enables @OP for Java by using special JavaDoc tags. This use of JavaDoc tags for attributes formed the original ideas for Java 5 annotations. The program code and the JavaDoc tags are reified as a Java Abstract Syntax Tree (AST) using Qdox[2]. Based on Generama[3], XDoclet uses standard template engines such as Velocity[4] and Freemarker[5] for generation of text-oriented output, and Jelly[6] for XML output. The generation process is driven by the template file, which consults the Java AST of the program code to feed the required information. The function of XDoclet is to seed the generation contexts for these template engines.

XDoclet is supported by XDoclet plugins, which provide task-specific generation functionality. XDoclet parses the handwritten program code and generates many artifacts such as XML descriptors and program code from it. These files are generated from templates that use the information provided in the program code and its JavaDoc tags.

Fraclet: An XDoclet plugin. To use XDoclet as a generation engine for Fraclet, it is necessary to define an XDoclet plugin. Basically, an XDoclet plugin is composed of two parts: *annotations* and *generators*. The annotations defined in XDoclet are special JavaDoc tags appearing in the program code as `/** @Attribute */`. These annotations enhance the program code with the metadata that is required by the component model but cannot be expressed in the Java code. The use of XDoc tags as annotations does not require any dependency towards a particular library. The Fractal attribute controller interface and component glue are generated using two Velocity templates. The primitive and abstract composite definition generators are implemented as Jelly templates.

The execution of the attribute controller and component glue generators results in the class diagram depicted in Figure 11. The class `FcClient` (resp. `FcServer`) extends the content class `Client` (resp. `Server`) to inherit from the business code of the component. The class `FcClient` implements the interface `BindingController` specified by the Fractal component model to allow its client interfaces to be bound to any compatible server interfaces. The class `FcServer` implements the interface `ServerAttributeController` generated by the attribute controller generator to support the dynamic reconfiguration of the component attributes. The interface `ServerAttributeController` must extend the `AttributeController` as specified in Fractal.

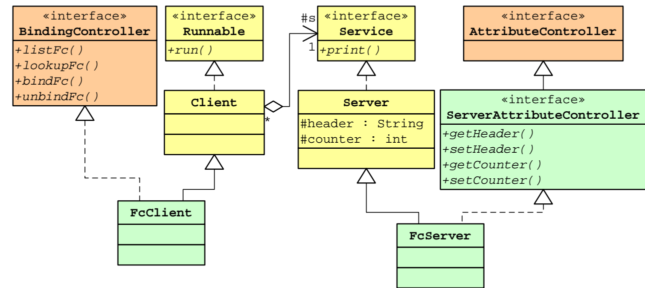

Figure 11. XDoclet implementation: The UML class diagram.

The execution of the generators of the primitive and abstract composite definitions results in the file architecture depicted in Figure 12. The primitive definition Server extends the abstract definition Service to declare the list of provided interfaces. The primitive definition Client is used by the abstract composite definition ClientComp to define the client component contained in this composite. The abstract composite definition ClientComp refers also to the abstract definition Service to resolve the dependency of the client component.

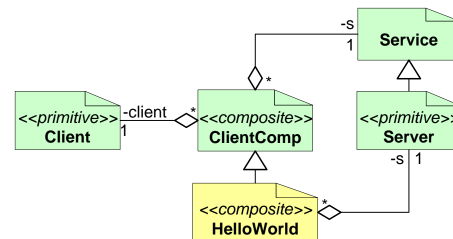

Figure 12. ADL descriptors.

5.2 Implementing Fraclet with Spoon

Spoon overview. Spoon is a Java 5 open compiler built on top of Javac [12]. Spoon provides the user with a representation of the Java AST in a metamodel, which allows both for reading and writing. Using this metamodel and a specific API, Spoon allows the programmer to process Java 5 programs. This processing is implemented with a visitor pattern that scans each visited program

[1] The XDoclet project: `http://xdoclet.codehaus.org/`

[2] The QDox project: `http://qdox.codehaus.org/`

[3] The Generama project: `http://generama.codehaus.org/`

[4] The Velocity project: `http://jakarta.apache.org/velocity/`

[5] The Freemarker project: `http://freemarker.sourceforge.net/`

[6] The Jelly project: `http://jakarta.apache.org/commons/jelly/`

element and can apply some user-defined processing jobs called *processors*. In particular, the processing can be annotation-driven, contrary to XDoclet.

Taking advantage of Java 5 features, Spoon also natively provides a framework that allows for the definition of code templates in pure Java. By specifying templates in pure Java, programmers can write them in their favorite Java IDE and benefit from all the advantages that come with it (incremental compilation, completion, syntax highlighting, contextual help, refactoring, wizards, etc.).

Fraclet using Spoon templates and processors. Our second implementation of Fraclet uses Spoon templates and processors to transform the Fractal base program enhanced with Java 5 annotations, and to generate some artifacts such as XML files for Fractal ADL. The main originality of this implementation compared to the XDoclet one is that the base program can be directly transformed by manipulating the metamodel of the base program. As explained in the previous paragraph, Spoon builds a metamodel of the base program (a representation of the Java AST) which allows the reading and writing of program elements. Thus, in this implementation of Fraclet, features can be directly injected into the original classes instead of extending them using inheritance. For example, the getter and setter methods related to the annotation `@Attribute` (see attribute controller generator in Section 4.2) are injected into the original class. As a matter of comparison, Figure 11 presents the class hierarchy obtained with the XDoclet implementation and Figure 13 shows the class hierarchy obtained with the Spoon implementation. In the first case, new classes `FcServer` and `FcClient` are generated to provide the wanted artifacts, whereas in the second case, artifacts are directly injected into original `Client` and `Server` classes.

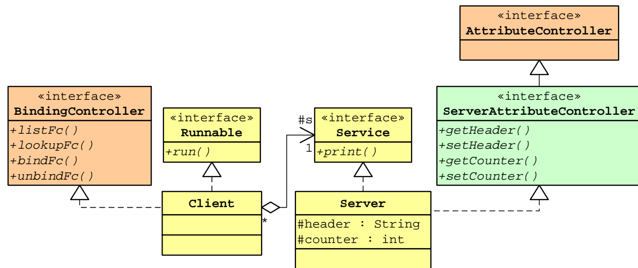

Figure 13. Spoon implementation: The UML class diagram.

6. Evaluation

This section aims at evaluating the benefits of using @OP for the development of Fractal applications.

Table 2 provides an empirical measure done on the application HelloWorld introduced in this paper. This evaluation confirms the statements claimed in Section 3 because it appears that the use of Fraclet saves 57% of the program code when considering the number of lines of Java source code and 88% of the ADL definitions.

Program Code	Unit	Fractal A	Fraclet B	Gain $G = A - B$	Rate G/A
Java	files	4	3	1	25 %
ADL	files	3	1	2	66 %
Java	lines	56	24	32	57 %
ADL	lines	26	3	23	88 %
Source	bytes	36 K	24 K	12 K	33 %

Table 2. Empirical measure performed on HelloWorld.

Table 3 provides an empirical measure done on the Comanche web server [7]. This application does not define any attribute and uses only three components with client interfaces. Therefore the overhead of the Fractal component model appears to be limited but the result of the evaluation makes appear that 41% of the Comanche source code can be saved.

Program Code	Unit	Fractal A	Fraclet B	Gain $G = A - B$	Rate G/A
Java	files	12	12	0	0 %
ADL	files	19	6	13	68 %
Java	lines	254	173	81	32 %
ADL	lines	88	38	50	57 %
Source	bytes	124 K	72 K	52 K	41 %

Table 3. Empirical measure performed on Comanche Web Server.

7. Related Work

This section compares our Attribute-Oriented Programming approach to the Generative Programming and Aspect-Oriented approaches already applied in the context of the Fractal component model.

7.1 Generative Programming

Integrated Development Environments (IDE), such as Eclipse, already provide program code generators to automatically fill some pieces of program code (*e.g.*, getter and setter methods for a given attribute), but these generators are static and limited. In particular, they are limited to the program code generation and provide only trivial generators. They do not provide a way of either generating methods combining several attributes or external artifacts (*e.g.*, descriptors, configuration files).

Similarly, the Fractal GUI tool is an IDE dedicated to the development of Fractal applications. It allows the developper to graphically design the components of an application. Once the application is fully described, Fractal GUI provides Java code generators to produce automatically skeletons of the component code —*i.e.*, components without business method code. Nevertheless, if the program code evolves once generated, Fractal GUI is not able to retrieve the evolution of the program code to integrate them.

Fraclet provides a solution to support continuous integration of Fractal non-functional code in the business code. The component artifacts are automatically generated from the current version of the program code. This approach supports either trivial generators (*e.g.*, getter/setter methods), or advanced generators (*e.g.*, binding controller), or external artifacts (*e.g.*, ADL definitions) using a small set of program code annotations. This approach is also complementary to IDEs approaches by providing reverse engineering capabilities to the program code generated by tools such as Fractal GUI.

7.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) aims at modularizing crosscutting code into systems [9]. It is motivated by the realization that there are concerns not well captured by current programming techniques. To overcome this issue, AOP proposes artifacts to modularize them by the use of advising and code introduction (inter-type declaration in AspectJ). The *advising* term refers to the traditional pointcut/advice model of AOP. Advices implement the crosscutting concern which is then woven to the base system by use of pointcuts.

[7] The definition of the Comanche web server is available at the following URL: http://fractal.objectweb.org/tutorial/index.html

A pointcut is able to pick out a set of join points (point in the execution of a program) where the advice codes apply. The *introduction* term refers to another feature of AOP, which allows the injection of new features into the base code, namely introducing a new field in a class, or it can force a class to implement a particular interface along with an implementation of that interface.

In AOKell, an implementation of the Fractal component model in Java, which uses AspectJ advising and introduction to implement the level of control of components [15], introduction is used to inject the implementation of control interfaces into content classes of components. Advising is used to implement some control interfaces, which require interception of interface calls such as the life cycle control interface which has to block calls on stopped components. Similarly to Fraclet, AOKell is able to inject the binding control interface implementation into components. Although the introduction mechanism seems perfectly well suited for this task, one major feature is missing. Indeed, with introduction the implementation of an interface can be forced, but the interface itself cannot be generated. It has to be provided by the aspect programmer. Thus, AOP introduction mechanism can only inject new features into existing classes but cannot generate new entities such as interfaces or classes from scratch. Moreover Fraclet is not tightly coupled to a particular implementation of Fractal. This means that Fraclet can be used either with AOKell or Julia [2].

The work presented in [14] proposes an annotation toolkit that allows building DoS resistant component-based systems. The proposed mechanism handles the robustness concern as separated concern. This concern is identified on the component interfaces at the architecture level, making it transparent to the developer of a component. Therefore, the annotations defined in this work are not used in the source code of the component but are applied in an *a posteriori* approach using the Fractal ADL tool. While bringing annotation benefits to the architecture level of application, this approach, based on AspectJ, is limited to the injection feature and does not support the generation of external artifacts. The objective of Fraclet is to facilitate the development of Fractal applications, while providing an easy and abstract syntax for programming the component's content. This approach hides the technical specificities of the Fractal component model to the developer and generates automatically most of the artifacts required by the model (API, descriptors). Moreover, Fraclet does not require any modification of existing Fractal tools.

8. Conclusion and Perspectives

This paper introduced the Fraclet annotation framework. We have shown that Fraclet makes the development of Fractal applications easier by factorizing redundant information and non-functional code, which is possible through the use of annotations. These annotations are dedicated to the Fractal component model specificities —*i.e.*, interface, binding, and controller. The associated generators interpret the annotated program code to produce various kinds of artifacts (*e.g.*, Java code, XML definitions). This approach allows the developer to save 50% of the traditional Fractal program code by hiding the implementation of the non-functional code. Two implementations of Fraclet are available: (1) using XDoc annotations and the XDoclet generation engine and (2) using Java 5 annotations and the Spoon generation engine.

The perspectives on this work enclose the extension of Fraclet to support tools such as the Monolog logging framework[8] and the Fractal Explorer tool. Monolog provides logging facilities to Fractal components. The goal of Fraclet will be to make easier the use of the loggers by generating either the configuration file required by the Monolog framework and the piece of code used to initialize a Monolog logger. Fractal Explorer is a graphical administration console that allows users to dynamically interact on their applications. Fractal Explorer allows the developer to customize the graphical user interface with the definition of an explorer plugin. An explorer plugin encloses the definition of a set of icons, panels, and actions for a given application. Nevertheless, the development of such a plugin can become a fastidious task for most of the developers. Thus, the goal of Fraclet will be to make easier the personalization of the Fractal Explorer thanks to a set of dedicated annotations. These annotations will take in charge either the generation of the explorer configuration file and the generation of some trivial actions. Another perspective for Fraclet is its integration in the Fractal GUI tool. This integration would ensure the backward compatibility between the program code and the graphical representation of the architecture provided by Fractal GUI when the program code evolves.

Availability. Fraclet is freely available under an LGPL licence at the following URL: `http://fractal.objectweb.org/`

[8] Monolog project: `http://monolog.objectweb.org/`

References

[1] ABDELLATIF, T. Enhancing the Management of a J2EE Application Server using a Component-Based Architecture. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'05)* (Porto, Portugal, August 2005), pp. 70–79.

[2] BRUNETON, E., COUPAYE, T., LECLERCQ, M., QUÉMA, V., AND STEFANI, J.-B. An Open Component Model and Its Support in Java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE'04)* (Edinburgh, UK, May 2004), vol. 3054 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 7–22.

[3] BRUNETON, E., COUPAYE, T., AND STEFANI, J.-B. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of 7th International Workshop on Component-Oriented Programming (WCOP'02)* (Malaga, Spain, June 2002).

[4] DEMICHIEL, L., AND KEITH, M. *Enterprise JavaBeans (EJB) Specification*, 3.0 ed. Sun Microsystems, Inc., Santa Clara, California, U.S.A, December 2005. `http://java.sun.com/products/ejb/`.

[5] ECMA INTERNATIONAL. *C# Language Specification*, 3.0 ed. Geneva, Switzerland, June 2005.

[6] FASSINO, J.-P., STEFANI, J.-B., LAWALL, J. L., AND MULLER, G. Think: A Software Framework for Component-based Operating System Kernels. In *Proceedings of the USENIX Annual Technical Conference, General Track* (Monterey, California, USA, June 2002), pp. 73–86.

[7] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification, Third Edition*. Addison-Westley Professional Computing, Santa Clara, California, USA, December 2005. `http://java.sun.com/j2se/1.5.0/docs/`.

[8] IBM CORPORATION. *SCA Service Component Architecture*, 0.9 ed., November 2005. Client and Implementation Model Specification.

[9] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)* (June 1997), vol. 1241 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 220–242.

[10] LECLERCQ, M., QUÉMA, V., AND STEFANI, J.-B. DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware. *IEEE Distributed Systems Online 6*, 9 (September 2005), 1–12.

[11] MEDVIDOVIC, N., AND TAYLOR, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering 26*, 1 (January 2000), 70–93.

[12] PAWLAK, R. Spoon : Annotation-Driven Program Transformation - The AOP Case. In *Proceedings of the 1st International Middleware Workshop on Aspect-Oriented Middleware Development (AOMD'05)* (Grenoble, France, November 2005), vol. 118 of *ACM International Conference Proceeding Series*, ACM Press, pp. 1–6.

[13] ROUVOY, R., SERRANO-ALVARADO, P., AND MERLE, P. Towards Context-Aware Transaction Services. In *Proceedings of the 6th International Conference on Distributed Applications and Interoperable Systems (DAIS'06)* (Bologna, Italy, June 2006), Lecture Notes in Computer Science, Springer-Verlag. To appear.

[14] SCHIAVONI, V., AND QUÉMA, V. *A Posteriori* Defensive Programming: An Annotation Toolkit for DoS-Resistant Component-Based Architectures. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06)* (Dijon, France, April 2006), ACM Press. To appear.

[15] SEINTURIER, L., PESSEMIER, N., DUCHIEN, L., AND COUPAYE, T. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)* (Stockholm, Sweden, June 2006), Lecture Notes in Computer Science, Springer-Verlag. To appear.

[16] WADA, H., AND SUZUKI, J. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)* (Montego Bay, Jamaica, October 2005), vol. 3713 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 584 – 600.

[17] WALLS, C., AND RICHARDS, N. *XDoclet in Action*. In Actions series. Manning Publications, December 2003.