Using Metadata in Aspect-Oriented Frameworks

Eduardo M. Guerra
Instituto Tecnológico da Aeronáutica (ITA)
Computer Science Division
guerraem@gmail.com

Jefferson O. Silva
Instituto Tecnológico da Aeronáutica (ITA)
Computer Science Division
jefferson.o.silva@uol.com.br

Fábio F. Silveira
Federal University of São Paulo (UNIFESP)
Technology and Science Department
fsilveira@unifesp.com.br

Clóvis T. Fernandes
Instituto Tecnológico da Aeronáutica (ITA)
Computer Science Division
clovistf@uol.com.br

Abstract

A traditional approach for building aspect-oriented frameworks has difficulty in modularizing crosscutting concerns that have different behaviors in diverse situations. Each crosscutting behavior needs to be implemented by an advice, which can lead to a combinatorial explosion in the number of advice. Intercepted classes can use metadata to adapt its behavior to these diverse situations. This paper presents an approach that merges aspect orientation and metadata creating a more flexible and extensible solution that results in a smaller number of advice.

Keywords aspect; framework; metadata; @OP

1. Introduction

An important characteristic of software frameworks is their ability to promote reuse. In fact, they can be defined as collections of classes that make up a reusable design for a specific class of software [7]. Basically, their reuse consists of capturing a common design for an application specific domain. Other important features are extensibility, modularity, and inversion of control (IoC). IoC means that the control flow shifts from the application to the framework.

Aspect-oriented programming (AOP), due to its crosscutting capability, opens a new research line with respect to reuse, extensibility and modularity in the development of applications. Aspect-oriented framework (AOF) aims at the same purpose of an object-oriented framework, however, an AOF counts on the AOP composition mechanisms, which include aspects, beyond classes [3] [4]. AOP composition mechanisms augment modularization of applications, since they allow encapsulation of crosscutting concerns [3]. There are papers that analyze modularization of functional concerns [11] [5] and non-functional ones [1] [12].

This work discusses the advice combinatory explosion that may occur with the utilization of aspect-oriented techniques. It also proposes metadata usage in intercepted classes to allow aspects to have behavior variations in a single crosscutting concern.

This paper proposes a structure that allows the expansion in the metadata schema, providing more flexibility in frameworks. In order to validate the presented ideas, it presents a case study that uses the technique addressed in this research.

This paper is outlined as follows. Section 2 details the complexities involved in the building of frameworks using aspect-oriented approaches. Section 3 discusses the concepts of metadata in an AOF and how they can help to reduce complexity. Section 4 presents the Metadata-based Logger, which is an AOF developed to exemplify the concepts in this paper. Section 5 analyzes a solution proposed by Camargo and Masiero [3]. Section 6 summarizes this research, outlining the main contributions and possible extensions.

2. Aspect-Oriented Frameworks

Consider these requirements for building a framework that logs method calls: calls to client methods must be divided by the logger in method name, declared exceptions, thrown exception, arguments, declared parameters, declared method return type and the actual method return. Each method call may be logged with any combination of the method parts previously mentioned. For instance, the framework could log only the method name for one method call, while it could log the whole method signature for another one. In addition to that, the framework must allow logging in two different locations: a database and a file. Analogously, it may log in one of the two locations or in both of them.

The conventional aspect-oriented solution for the above requirements typically implements the method parts and the log locations as regular object-oriented classes. Since logging is a crosscutting concern, it is usually modularized by aspects.

The framework aspects need to include logging references inside advice. This entails a high syntactic coupling relationship [14] among advice and the framework classes that implement the logging.

Figure 1 depicts a UML class model representing the relationship among advice, the logging implementation classes and the client code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGPLAN'05 June 12–15, 2005, Location, State, Country.
Copyright © 2004 ACM 1-59593-XXX-X/0X/000X…$5.00.

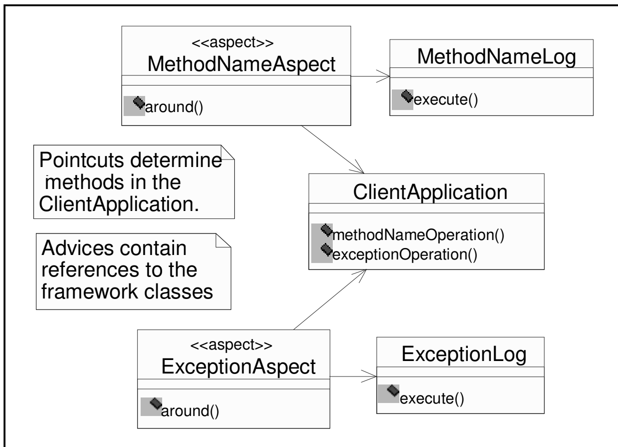

Figure 1. Aspects, framework classes and the application.

The logger framework contains only one crosscutting concern, which is to capture method call information from client methods and send them to the logging implementation classes. However, two advice are needed for capturing the information of the two different client methods.

The main reason for having two advice is that both of them have a reference to a framework class. Thus, in Figure 1, for collecting a method name it is necessary to use the aspect *MethodNameAspect*. Similarly, for collecting the method declared exceptions, it is necessary to use the aspect *ExceptionAspect*. It means that, using the AOP approach, a crosscutting concern implemented by an advice cannot be reused for different framework logging classes.

A characteristic that can vary in one crosscutting concern is called variability. In the conventional aspect-oriented technique, the number of variabilities is directly proportional to the number of advice.

Figure 2 displays the number of framework variabilities and the number of advice. This relationship can be represented by the function $f(x) = x$.

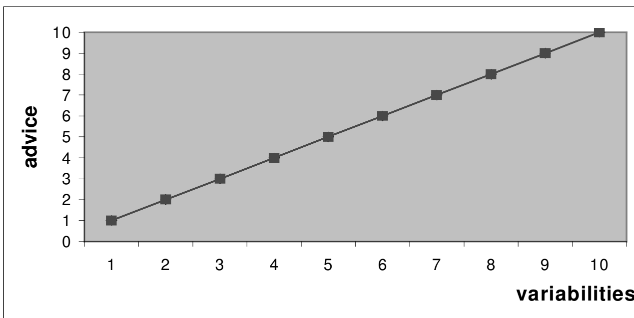

Figure 2. Relationship among advice and variabilities

The function $f(x)$, portrayed in Figure 2, does not consider any variability combinations. Some business rules cases specify that variabilities can be combined. For instance, in the framework cited previously, combining the database and file variabilities would cause the framework to log in both locations, implying in the definition of a new behavior. That is, a new advice would have to be created, implementing the crosscutting behavior that allowed the logging to be performed in both locations.

When it is considered the possibility of variability combinations, it is necessary to redefine the function that relates the number of advice and variabilities. The new function can be determined by the fundamental principle of counting, which regards combinations. Thus, it is defined as follows: $g(x) = 2^x - n$, where x is the total framework variabilities and n is the number of variabilities that must be present. For instance, in the logger framework at least one method call part must be captured for logging and at least one location must be chosen for the log information to be registered. Therefore, for the logger framework case $n=2$.

Figure 3 illustrates the advice growth scale.

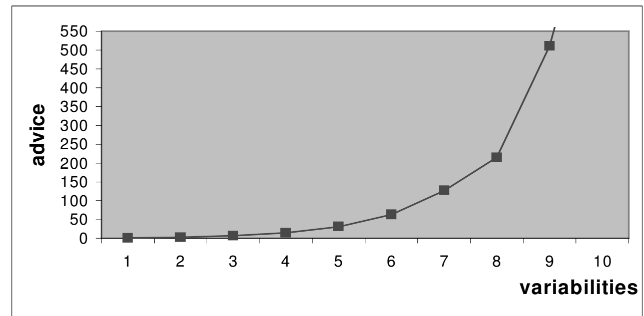

Figure 3. Advice and variabilities with combinations

When variability combinations are considered, the number of advice that needs to be created grows exponentially. The number is significant even for a little variability.

The extensibility is affected as well. Suppose that an application needs the framework to support another variability. In the best scenario, which is the case of variabilities that cannot be combined (Figure 2), a new advice must be created. In the worst one, which is the case of variability combinations (Figure 3), the number of advice that must be created is defined as follows: $g'(x) = g(x) + n$, where x is the total framework variabilities and n the number of variabilities that must be present. The function $g'(x)$ can be factored into: $g'(x) = 2^x$.

3. Aspect-Oriented Frameworks Based on Metadata

Niso [9] defines metadata as structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. This paper defines frameworks that utilize metadata as a basis for decision-making in runtime as metadata-based frameworks (MBF).

In the Java language, with the introduction of JSR 175 [8], it became possible the usage of attribute-oriented programming (@OP). Attribute-oriented programming is a program-level marking technique that allows the programmers to mark program elements (e.g. classes and methods) to indicate that they maintain application-specific or domain-specific semantics [13].

Some frameworks, as EJB 3.0 [6] and Hibernate [2] for example, make extensive use of @OP. EJB 3.0, for example, uses metadata, defined in annotations or XML files, for the definition of crosscutting concerns variabilities like transaction management and access control. As EJB 3.0 is a specification, it does not define how the behavior must be implemented.

The difference between @OP and MBFs is purely conceptual. While @OP focuses on coding (with annotations), independently of the manner metadata will be processed, MBFs focus on metadata processing in runtime, irrespectively of the way they are stored.

A convenient fashion to solve the syntactic coupling drawback [14], detailed in section 2, is to use metadata for the definition of crosscutting concern variabilities in an AOF. The technique consists of a two-step activity: (i) remove all references to variabilities from advice; and (ii) include the necessary information for the composition activity in metadata.

Metadata usage prevents the combinatory explosion that may result as a consequence of the previously mentioned coupling, creating a more effective modularization. New behavior defined by variability combinations can be configured in metadata, dispensing with the need for the creation of new advice. Therefore, a single crosscutting concern can support many variabilities.

Neto et al [14] present a definition for syntactic and semantic coupling. Moreover, Yang and Tempero [15] have shown criteria for dealing with indirect coupling. The use of metadata helps to mitigate the syntactic coupling, however, it does not eliminate all kinds of semantic or indirect couplings.

This approach also allows the insertion of new functionality in the framework by extending the metadata schema without having to create additional advice.

4. Metadata-based Logger Framework

Metadata-based Logger [10] is an aspect-oriented framework based on metadata, which we developed to comply with the requirements described for the framework in section 2. It uses metadata to indicate what method call information must be logged and the locations where the log must be recorded. The framework collects attributes through metadata, which can be stored in annotations or in an XML file. For instance, a client method annotation could specify that an aspect must log the method name in the database and file. Another annotation could determine that the name and the declared exceptions of a method must be logged only in the database.

Implementing Metadata-based Logger framework using the usual aspect-oriented techniques would lead to an explosion in the number of advice, since there would be numerous combinations.

Metadata should be placed prior to client methods declarations when annotations are used. In the case of an XML file, it must conform to the framework XML syntax. Metadata configure which method call parts will be collected and where the logging will take place.

Figure 4 shows a client code snippet using annotations to specify to the framework the logging information for a method.

```
@LogMarker(
    logInfo = { LogKeys.METHOD,
                LogKeys.PARAMETER,
                LogKeys.ARGUMENT,
                LogKeys.DECLARED_RETURN_TYPE,
                LogKeys.RETURN_TYPE,
                LogKeys.EXCEPTION,
                LogKeys.THROWN_EXCEPTION },
    logLocation = {
                LogKeys.FILE,
                LogKeys.DATABASE }
)
public void testLog(Integer i) throws NullPointerException, IOException {
    System.out.println("Method executed");
}
```

Figure 4. Client method configured with an annotation

Metadata-based Logger utilizes a flexible architecture. The *LogKeys* interface contains the prefixes of the classes that implement the variabilities. Each variability has a correspondent suffix. The keys put in the *logInfo* attribute are suffixed with *LogInfo* to form the name of the variability class, while keys put in the *logLocation* attribute are suffixed with *Location*. For instance, a class that implements the logging in a file is named with the combination of the prefix defined in *LogKeys* and the correspondent suffix, resulting in the name *FileLocation*.

Analogously, Figure 5 portrays the same previous configured method following the XML syntax. The variability name formation rules are the same detailed for annotations.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<logmarker>
   <method signature="void logging.LoggerApplication.testLog(Integer)">
       <info>
           <value>Method</value>
           <value>Parameter</value>
           <value>Argument</value>
           <value>DeclaredReturnType</value>
           <value>ReturnType</value>
           <value>Exception</value>
           <value>ThrownException</value>
       </info>
       <type>
           <value>DB</value>
           <value>File</value>
       </type>
   </method>
</logmarker>
```

Figure 5. Method configured with the XML syntax

The framework advice are responsible for capturing the common crosscutting behavior, which in the case of the logger framework consists of mapping method calls information and delegate the rest of the process to a factory that is able to interpret the specified metadata. For increasing the expressiveness of the pointcuts, the aspect that encapsulates the crosscutting behavior was made abstract (`AbstractAspectLogger`).

The framework factories contain logic for metadata consumption. They interpret the defined metadata for each method and recover the variability prefix. In this manner, the factories can instantiate the correct class. It provides more extensibility because if a new variability is created, all that is required is to create the class that implements the functionality and define its prefix in *LogKeys*. The Logger implements a structure that uncouples the metadata consumption logic from the component logic and also allows many kinds of metadata storage. Figure 6 depicts a UML class diagram that abstracts the architecture of the framework.

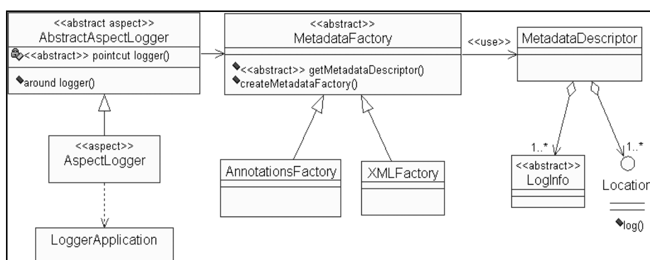

Figure 6. Metadata-based Logger UML class diagram

For comparison sake, if the traditional aspect-oriented approach have been used to implement Metadata-based Logger framework, we would needed a plethora of advice to represent all the behavior combination possibilities. The Metadata-based Logger framework has a total of ten variabilities. The function *g(x)*, demonstrated in section 2, calculates the number of advice that need to be created for a specified number of variabilities. For every combination made, there must be at least two variabilities: one for collecting data for the logging; and another one for choosing a location for the logging to take place. It can be verified that it would be necessary 510 advice to cover all possible combinations.

5. Table-based Calculation Framework Revisited

This section introduces the Table-based Calculation, a framework that utilizes aspect-oriented techniques for treatment of business rules [3]. The aim is to suggest a metadata-based solution for the framework.

Camargo and Masiero [3] present a crosscutting framework that modularizes a business rule that has the objective to increment or decrement intercepted values from the base code. A percentage calculation is performed on the captured value according to its range. The authors provide two business rule examples: the Meal Ticket, as a rule without reduction; and Income Tax, as a rule with reduction.

There are variations on the manner these values can be obtained. Table-based Calculation provides composition alternatives, which are implemented in abstract aspects that contain the logic for the extraction of values from the base code.

The necessity for composition alternatives are justified because a value to be captured may be in many places such as: in the method return; in the arguments of a method; in a local variable of the calling object (*this*); or in a local variable of the target object (*target*).

The authors present the concretization of an aspect that implements the Meal Ticket business rule (Figure 7).

```
public pointcut obtainValue():
   call (* BaseSalary.getValue())
   && withincode (* Employee.calculateMySalaryBasedOnEvents(int, int));
public String getNameOfTableClassWithPackage(){
      return "tableBasedCalculation.instantiation.MealTicket";
}
public String getValueType(){ return "float";}
```

Figure 7. Meal Ticket implementation aspect [3]

The pointcut `obtainValue()` determines the method where the aspect is going to act. The method `getNameOfTableClassWithPackage()` indicates the class of the framework that implements the business rule. Similarly, the method `getValueType()` comprises information about the method return type. As discussed previously, the syntactic coupling among advice and framework classes that implement business rules imply in the creation of many advice.

In Figure 7, the high coupling occurs because of the methods `getValueType()` and `getNameOfTableClassWithPackage()`. If another business rule (e.g. Income Tax) needs to be created there will have to be another aspect which the only difference will be the referenced class.

References to the framework classes do not need to be contained inside the aspect. Base code methods could have been annotated or described in XML to contain these references. Figure 8 illustrates how the client method could be annotated. Although it is used an annotation as metadata, any other type could have been used.

```
@Calculation(
    tableClass = {"MealTicket"},
    valueType = {"float"}
)
public float getValue() {
    return value;
}
```

Figure 8. Annotated method

Metadata-based solutions face a limitation on the base code value extraction. It is possible that the value needed from the base code is in a method local variable. There is a limitation for Java (until version 6) that does not provide a reflection mechanism for obtaining local variables, making it impossible to capture these types of values in runtime. For the specific case of annotations, it is possible to annotate variables, however they will only serve as documentation. For this case, the traditional aspect-oriented approach would have to be used. That is, it would be necessary to use composition alternatives including references to variabilities in the framework aspects.

6. Conclusion

This paper has proposed the utilization of metadata in Aspect-Oriented Frameworks (AOF). Conventional aspect-oriented solutions accomplish all the composition activity inside aspects. The primary consequence of this approach is that advice need to reference the framework classes that implement the business rules. A characteristic that can vary in the same crosscutting concern is called a variability. This syntactic coupling [14] results in the growth of the number of advice directly proportional to the number of variabilities.

Some business rules require the combination of framework variabilities. Variability combinations determine new behavior. When combinations are considered, the scale of the number of advice grows exponentially.

Metadata usage in AOFs breaks the existing syntactic coupling [14] among advice and variabilities, resulting in more effective modularization. References to variability implementation classes can be included in XML files, in annotations or in any other type of metadata.

In this way, metadata can augment reuse and extensibility in AOFs. A crosscutting concern modularized by an advice can be reused for many variabilities. Moreover, it has been verified that the inclusion of a new variability requires less effort when compared to customary aspect-oriented solutions.

The technique presented in this paper, which merges aspect orientation and metadata, is especially interesting for cases when variabilities need to be combined. However, this paper does not make an exhaustive analysis of the possible situations in which the uses of metadata are applicable. An extension to this work would be to analyze different use cases, as the analyses of other use situations, as well criteria for the choice between the traditional aspect-oriented approach and the metadata-based approach.

It is worth mentioning that it is a work in progress and the next phase of the research is to apply the concepts here introduced in a real application as a means to provide an estimate on the number of advice that needs to be created.

7. Bibliographic References

[1] Rashid, A. and Chitchyan, R. (2003) Persistence as an Aspect. In: Proc. of the 2nd International Conference on Aspect Oriented Software Development (AOSD) Boston–USA, March.

[2] Bauer, C. and King, G. (2005) Hibernate in Action. Manning Publishing Co.

[3] Camargo, V.V. and Masiero, P.C. (2005) Frameworks Orientados a Aspectos. In: Anais do 19º Simpósio Brasileiro de Engenharia de Software (SBES'2005), Uberlândia-MG, Brasil, October.

[4] Camargo, V.V., Ramos, R.A. and Masiero, P.C. (2004) Implementação de Variabilidades em Frameworks Orientados a Aspectos desenvolvidos em AspectJ. In: Relatório do 1º Workshop de Desenvolvimento de Software Orientado a Aspectos (WASP'04) – realizado em conjunto com o SBES'2004, Brasília, DF, Brazil, October.

[5] Cibrán, M., D'Hondt, M. e Jonckers, V. (2003) Aspect-Oriented Programming for Connecting Business Rules. In: Proc. of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA, June.

[6] DeMichiel, L. and Keith, M. (2005) Enterprise JavaBeans (EJB) Specification, 3rd ed.

[7] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[8] JSR 175. JSR 175 (2003) A Metadata Facility for the Java Programming Language. Available at: http://www.jcp.org/en/jsr/detail?id=175.

[9] Niso (2004) Understanding Metadata. Bethesda MD. Niso Press. Available at: http://www.niso.org/publications/press/UnderstandingMetadata.pdf.

[10] Metadata-based Logger (2008) Metadata-based Logger Framework Available at: https://sourceforge.net/projects/metadatalogger/, July.

[11] Suvée, D., Fraine, D.B. and Vanderperren, W. (2005) “FuseJ: An Architectural description language for unifying aspects and components”. In: Proc. of the 1st Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT’05), Chicago.

[12] Vanhaute, B., Win, B. and Decker, B. (2001) “Building Frameworks in AspectJ”. In: Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP), Separation of Concerns Workshop. pp. 1-6, June.

[13] D. Schwarz (2004) “Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5,” In ON Java.com, O’Reilly Media, Inc., June.

[14] Neto, A. C., Ribeiro, M. M., Dósea, M., Bonifácio, R., Borba, P. (2007) Semantic Dependencies and Modularity of Aspect-Oriented Software. In: 1st Workshop on Assessment of Contemporary Modularization Techniques, May

[15] Yang, H. Y., Tempero, E. (2007) Indirect Coupling as a Criteria for Modularity. In: 1st Workshop on Assessment of Contemporary Modularization Techniques, May