Aspect-Oriented Generation of the API Documentation for AspectJ

Michihiro Horie
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku,
Tokyo 152-8552, Japan
www.csg.is.titech.ac.jp/ horie

Shigeru Chiba
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku,
Tokyo 152-8552, Japan
www.csg.is.titech.ac.jp/ chiba

ABSTRACT

Through the development of a framework or a class library, writing the document on application programming interface (API) is essential. The document on the API, which we call *the API documentation*, is mainly read by programmers who want to develop their applications on top of that framework or library. In this paper, we present a tool named *CommentWeaver*, which generates the API documentation of a framework/library written in AspectJ. CommentWeaver extracts the descriptions for the API documentation from both classes and aspects in the source files, and then it *weaves* them for generating the API documentation in HTML. Although ajdoc similarly generates the API documentation for AspectJ, the generated API documentation does not directly present the exact behavior of the API if it is affected by aspects.

Categories and Subject Descriptors

D.3.3 [**Prgramming Languages**]: Language Constructs and Festures—*domain spepcific aspect languages*

General Terms

Languages

1. INTRODUCTION

Writing documents is a significant part of software development. Such documents include the results of requirement analysis and various levels of specifications of the software. In object-oriented programming, class diagrams will be written. Developers also have to write users' manual of the software. If the software is a framework or a class/function library, then the developers have to write the document on the application programming interface (API) of that framework or library.

The document on the API, which we call *the API documentation* in this paper, is read by user programmers who want to develop their applications on top of that framework or library. They read the API documentation to know how to write a program using that framework or library; the API documentation is its users' manual. Note that the API documentation is not comments or remarks for helping programmers understand the internal implementation of the framework or library. In the case of class libraries, thus, the API documentation usually does not include private methods or fields because they are not visible from the outside. It is the API documentation that describes what function is provided by each method or field exposed to the user programmers.

Since writing the API documentation together with a source program is known as a good practice, it is commonly seen in a few languages such as Common Lisp and Emacs Lisp. In Java, the API documentation is written as part of comments, known as *doc comments*, in a source program and then the javadoc tool [13] processes the source program and generates the API documentation in HTML. The syntax for writing the API documentation is standardized; the documentation must be surrounded with /** and */ and special tags such as @param can be used.

An aspect-oriented programming (AOP) language AspectJ [4] also follows this idea; it provides the ajdoc tool [15] similar to javadoc, which generates the API documentation with the same approach as javadoc. However, the generated API documentation is not satisfactory because the target language is an AOP language. As javadoc does, ajdoc first extracts the API documentation from the comments associated with language constructs, including classes, methods, aspects, pointcuts, and advices. Then ajdoc writes the extracted API documentation out in HTML files. When it writes out comments, it sorts out the descriptions and changes the format for better readability but it does not change the structure of the document. The structure of the document is the same as the program structure; the API documentation consists of class parts and aspects parts and these parts consist of their methods, fields, and advices. This is not suitable for representing detailed specifications of the API.

For example, if an advice affects the behavior of a method, the aspect part describes the effects of the advice whereas the class part separately describes the original behavior of the method before the advice is applied. The method behavior after the advice is woven is not clear. While the documentation pages include clickable links between methods and advices advising those methods, the users of the framework or library cannot understand at a glance the ex-

DSAL'09, March 3, 2009, Charlottesville, Virginia, USA.

act behavior of the methods that they are going to use. They have to click the link and consider by themselves how the advice modifies or extends the behavior of that method. Although this link would be useful for the programmers who are extending the framework or library and thus are interesting in the internal implementation, it is not appropriate for the users who just want to *use* the framework or library through the well-documented API. Furthermore, clicking the link may not reveal the exact behavior of the method if the method internally calls another method that is also modified by another advice.

To address this problem, this paper presents our javadoc-like tool for generating the API documentation of a framework or a library written in AspectJ. Our tool generates the API documentation for framework/library users who write a program using that framework/library as a black box. Thus, our tool extracts the descriptions for the API documentation from both classes and aspects in the source files and then it *weaves* them before producing HTML pages. A pointcut language for controlling this weaving is also provided. The API descriptions are sorted out according to the corresponding entry points in the programming interface and thereby the document structure is different from the program structure. Since the program structure in AOP reflects the concerns of the framework/library implementers, we should choose a different structure for the documentation, which will be read by the users having different concerns. For example, if an advice affects the behavior of a method exported outside of the framework, the description of that advice is shown together in the API documentation with the description of the method. This is because, from the users' viewpoint, the method is an entry point in the framework API whereas the advice is not; it is recognized from the outside only through the behavior of that method. Furthermore, our tool does not generate an HTML page collecting all the descriptions related to one aspect unless the aspect is abstract and exposed to be inherited by aspects written by framework users.

In the rest of this paper, we first discuss problems of the existing AOP tools for generating API documentation in Section 2. In Section 3, we propose our javadoc-like tool for AspectJ. In Section 4, we present an example of the use of our tool. Then we have discussions about related work in Section 5. Section 6 concludes this paper.

2. PROBLEMS OF EXISTING TOOLS

Although AOP makes the implementation of libraries and frameworks more modular, their API documentation generated by the existing tools such as ajdoc tends to be unsatisfactory. This is due to the document structure that follows the program structure and thus does not respect the user programmers' concerns. In the program, crosscutting implementation concerns are separated but this separation makes it difficult for the users to understand how to use the framework or library.

2.1 Internally-used aspects

A library or framework implemented in AspectJ often includes aspects that are internally used for implementing crosscutting concerns such as transactions, exception handling, and logging. Such an aspect is never exposed to the users of the library or framework. The users should not have to see the existence of that aspect because this is an implementation detail.

```
public class Stack<E> extends Vector<E> {
        :
   /** Removes the object at the top of this stack and
    *  returns that object as the value of this function.
    *
    *  @return The object at the top of this stack
    *  (the last item of the <tt>Vector</tt> object).
    */
    public synchronized E pop() {
        E obj = peek();
        removeElementAt(size() - 1);
        return obj;
    }

   /** Looks at the object at the top of this stack
    *  without removing it from the stack.
    *
    *  @return the object at the top of this stack
    *  (the last item of the <tt>Vector</tt> object).
    */
    public synchronized E peek() {
        return elementAt(size() - 1);
}}

aspect StackChecking {
   /** @exception EmptyStackExcpetion if this stack is empty. */
   before(Stack s): execution(* Stack.peek()) && this(s) {
       if (s.size() == 0) throw new EmptyStackException();
}}
```

Figure 1: A revised Stack class by using an aspect

For example, we reimplemented in AspectJ the java.util.Stack class contained in the J2SE 5.0 library [12]. Figure 1 shows the program. The java.util.Stack class has the peek method that returns the top element of the stack but does not remove it. The peek method checks the pre-condition for the stack, which is that the size of the stack is not zero. If the size is zero, the method throws a runtime exception EmptyStackException. We separately implemented this check of the pre-condition by an aspect StackChecking.

Note that the doc comment about the possibility of throwing an EmptyStackException belongs to the before advice in the StackChecking aspect. This means that the API documentation generated by the existing tools such as ajdoc presents this description in the page of the StackChecking aspect. The page of the Stack class does not mention the possibility of throwing an EmptyStackException. This separation enforces extra work on the library users, who cannot understand the exact behavior of the peek method by looking at only the page of the Stack class. The users have to click the hyper-link on that page and also look at the page of the StackChecking aspect. This is not acceptable because the StackChecking aspect was written only for improving the modularity of the implementation and thus it is part of implementation details that should be invisible from the library users.

Furthermore, the API documentation never tells the users that the pop method may also throw an EmptyStackException because this method internally calls the peek method. In this case, the users cannot see this fact without looking at the source code of the Stack class. The API documentation does not provide a hyper-link representing that the StackChecking aspect indirectly affects the pop method. This is a serious problem; in fact, the original API documentation of the J2SE 5.0 library mentions that the pop method may throw an EmptyStackException.

2.2 A naive solution

A naive solution is giving up writing the API documentation for aspects in the source program of the aspects. When we write the doc comments for classes, we could also write the effects of aspects together within the doc comments for the classes. In the case of the StackChecking aspect in Figure 1, we could rewrite the doc comments for the peek and pop methods so that the doc comments directly present that the two methods may throw an EmptyStackException. Recall that, in Figure 1, this fact was written in the doc comments for the StackChecking aspect.

This solution makes the API documentation acceptable from the viewpoint of the framework/library users. On the other hand, it is not acceptable from the implementers' viewpoint. Since the effects of aspects depend on the implementation of the aspects, the doc comments about those effects should be in the source code of the aspects. This improves the maintainability of the aspects. This dilemma between the users and the implementers is due to the fact that they prefer different ways of decomposition. The implementers want to decompose according to implementation concerns while the users want to decompose according to functional concerns visible from the users.

Another drawback of the solution presented here is that the doc comments are fragile. Since we must manually write the doc comments about aspects in the source code of the classes affected by those aspects, we must update the doc comments whenever the pointcut definitions are modified. Enumerating the affected classes and methods is not a simple task. Furthermore, if the pointcut includes a wild card, the global analysis of the program is necessary for writing correct doc comments.

3. ASPECTS FOR DOCUMENTATION

To address the problem mentioned in the previous section, we proposes a new documentation system named *CommentWeaver* for AspectJ. This generates the API documentation mostly as ajdoc does. However, unlike ajdoc, it does *weave* the doc comments from classes and aspects before it writes out the HTML pages of the API documentation. It also provides special tags for controlling the weaving. CommentWeaver generates the API documentation similar to what we obtain by the naive solution proposed in Section 2.2 while it allows the implementers to write doc comments in the source code of the aspects. It weaves doc comments from classes and aspects and puts them together in the page of classes if needed. CommentWeaver does not generate a page for internally-used aspects whereas it generates for public aspects. The doc comments of the internally-used aspects are scatteringly shown in the pages of the public classes affected by those aspects.

3.1 Join points for CommentWeaver

The document processing by CommentWeaver can be explained as the execution of programs written in AOP languages. For AOP languages, Masuhara et al. proposes the ABX model [10, 9], which is represented by the formula $A \times B \rightarrow X$, where A and B are programs and X is the execution of the programs. In AspectJ, A is a set of classes and B is a set of aspects. When an AspectJ program is run, at each execution point in X, the language performs the computation specified by an element in A or B.

CommentWeaver is an AOP system explained by the ABX model. A is a set of doc comments written in classes and B is a set of doc comments in aspects.[1] X is the generation of the API documentation. At each execution point in X, CommentWeaver writes out a document taken from A or B. The execution points, *i.e.* join points, are:

- when the description of a class is written out,
- when the description of a field is written out, or
- when the description of a method is written out.

CommentWeaver provides a pointcut language. It is used for specifying the execution points at which doc comments from B is written out. The pointcuts are described by using special javadoc tags such as @caller in doc comments.

3.2 Pointcuts for doc comments

For simplicity, CommentWeaver provides two kinds of pointcuts: one is implicit and the other is explicit.

3.2.1 Implicit pointcut

By default, all doc comments in aspects have implicit pointcuts. Programmers do not have to write any other pointcuts if they are satisfied with the implicit ones. The implicit pointcuts select the execution point when CommentWeaver writes out the description of the entity, such as a method, that the corresponding advice affects. If a doc comment is for a named pointcut, it is processed as if it is the doc comment for the advice using that named pointcut.

For example, if the pointcut of an advice is execution, then the doc comment for that advice is written out as part of the description of the method selected by that execution pointcut. If the execution pointcut contains a wild card as following:

```
execution(void Figure.register*(..))
```

then the doc comments for that advice is included in the description of all the methods selected by that pointcut, *i.e.* all the methods in Figure starting with register.

Since CommentWeaver runs before runtime, it uses only the shadow [11] of the join points selected for advices. The cflow and if pointcuts are ignored. Only so-called accessor pointcuts call, execution, and set are considered (see Table 1). An abstract named pointcut, whose body is not defined yet, is also ignored.

3.2.2 Explicit pointcut

If the implicit pointcuts are not sufficient, programmers can explicitly specify pointcuts for doc comments. An advice of AspectJ affects not only the behavior of the selected join point but also indirectly the behavior of other methods in the call chain. For example, the advice in the StackChecking shown in Figure 1 affects not only the peek method that its execution pointcut selects. It also indirectly affects the pop method because it calls the peek method. The advice may affect the behavior of all the methods that directly or indirectly call the peek method.

[1] Another explanation is that A and B are programs producing the API documentation according to the doc comments in classes and aspects, respectively. The doc comment can be regarded as a self-printing program or something like a quoted expression in Lisp.

Table 1: The pointcuts of CommentWeaver

		Selects the execution points when CommentWeaver writes out the description of
implicit pointcut for	call	the caller methods
	get, set	the methods accessing the fields
	execution	the methods selected by execution
@caller		the methods directly/indirectly calling the method selected by the implicit pointcut
@callerclass		the classes directly/indirectly calling the method selected by the implicit pointcut
@callee		the methods called from the method selected by the implicit pointcut
@calleeclass		the classes of the methods called from the method selected by the implicit pointcut

To reflect the effects of aspects on the description of those methods in the call chain, CommentWeaver allows programmers to explicitly append execution points when doc comments are written out. Table 1 lists the pointcut for appending execution points. The @caller tag is a pointcut for including a doc comment in the description of the methods in the call chain, which directly or indirectly call the target method selected by the implicit pointcut. The @callerclass tag includes a doc comment in the description of the classes in the call chain. CommentWeaver uses the call chain obtainable by global static analysis. For example, if the @caller tag appears in a doc comment as following:

```
/** @caller()
 *    :      */
before(Stack s) : execution(* Stack.peek()) && this(s) { ... }
```

Then the comment following @caller() is included in the description of the methods that directly or indirectly calls the peek method. For example, the description of the pop method includes the comment for this advice.

The @callee tag is usually used with the call pointcut of AspectJ. Since the call pointcut of AspectJ selects the join points when a method m is about to call another method n specified by that call pointcut, the implicit pointcut of CommentWeaver selects not the callee method n but the caller method m. Thus, by default, the doc comment is appended to the description of the caller method m. If programmers want to append a document to the description of the callee method n, they have to use the @callee tag. For example,

```
/** @callee()
 *   :       */
before(Stack s) : call(* Stack.peek()) && target(s) { ... }
```

the doc comment following @callee() is included in the description of the callee method peek. Note that, if the @callee tag is not used, the doc comment for this advice is included in the description of the methods directly calling the peek method. The @callee tag can be used together with the @caller tag. The doc comment following this:

```
@caller() && @callee()
```

is appended to the description of both the caller and the callee methods.

The @caller tag and the @callee tag can take a parameter, which is either within or exec. These parameters restrict the range of the selected execution points. For example,

```
@caller(within(java.util.*))
```

the doc comment following this tag is included in the description of the caller methods within only the java.util package. It is not included in the description of the methods in the other packages even if those methods indirectly call the method selected by the implicit pointcut. The meanings of within is as following:

- within(⟨*Class Pattern*⟩)
  This selects only the classes that the ⟨*Class Pattern*⟩ matches and the methods declared in those classes.
  e.g. @callee(within(Figure+))
  This selects the callee methods in Figure or its subclasses.

If a method indirectly calls the method selected by the implicit pointcut, within and exec require that the entire call chain satisfies the condition. Suppose that a method m calls another method m_1, then m_1 calls another method m_2, ..., finally m_n calls the method m_{adv} selected by the implicit pointcut. If m is selected by @caller(within(R)), not only m but also m_1, ... m_n must be within R.

3.3 Advices for doc comments

While the @caller and @callee tags correspond to pointcuts, the text of the doc comments correspond to advices. By default, the text is appended to the description of the entity selected by the implicit or explicit pointcut. CommentWeaver processes the standard javadoc tags such as @exception and @return as javadoc does.

3.3.1 around *advices with an* execution *pointcut*

An around advice applied to the join points selected by an execution pointcut overrides the original computation by the method. Hence, the doc comment for that around advice is also substituted by CommentWeaver for the entire doc comment for the overridden method.

If the around advice calls proceed to execute the overridden method, the doc comment for that method should reflect this fact. To do that, programmers can write @proceed in the doc comment for the around advice. The @proceed tag is replaced with the doc comments for the overridden method. For example,

```
/** @proceed
 *  @exception EmptyStackException if this stack is empty.
 */
Object around(Stack s) : execution(* Stack.peek()) && this(s) {
  if (s.size == 0) throw new EmptyStackException();
    return proceed(s);
}
```

since this around advice overrides the peek method in Stack, the description of the peek method in the API documentation is taken from the doc comment for this around advice.

However, @proceed in the doc comment is replaced with the original doc comment for the peek method.

4. EXAMPLES REVISITED

The problems presented in Section 2 can be solved by CommentWeaver. In Section 2.1, we presented the StackChecking aspect and a problem of its API documentation, in which the effects of the before advice are not reflected on the description of the pop method. This problem is solved if we modify the doc comments on the before advice as the following:

```
/** @caller() && @callee()
 *     @exception EmptyStackException if this stack is empty.
 */
before(Stack s) : execution(* Stack.peek()) && this(s) { ... }
```

The doc comment will be appended not only the description of the callee method peek but also the descriptions of the caller methods such as pop, which directly or indirectly calls the peek method.

5. RELATED WORK

The languages such as WEB [6] and CWEB [8] are based on the concept of *literate programming* [7], which promotes better documentation of programs. WEB consists of two languages, that is, TEX for writing documentation and Pascal for programming. In the WEB source files, programmers can write a Pascal program with the TEX text for improving the readability of the Pascal program. The operation of *WEAVE* extracts the TEX text from the WEB source files and then it generates the .tex files for the documentation. CommentWeaver also promotes better API documentation by improving the readability of a program with respect to the crosscutting structures of aspects.

The Aspect-Aware Interface (AAI) [5] is a novel interface for AOP to address the obliviousness [2] property in AOP programs. The API documentation generated by CommentWeaver can be regarded as one presentation of the AAI although the AAI is a programming-language concept but CommentWeaver is a tool for generating API documentation.

6. CONCLUDING REMARKS

This paper presents a novel documentation system named *CommentWeaver*, which generates the API documentation of a framework/library written in AspectJ. CommentWeaver extracts the descriptions for the API documentation from both classes and aspects in the source files, and then it *weaves* them for generating the API documentation in HTML.

CommentWeaver is an AOP system that can be explained by the ABX model. As other AOP languages, CommentWeaver also provides a pointcut language, which specifies the execution points at which doc comments is written out as part of the API documentation. This pointcut language is used for specifying the execution points that are indirectly affected by aspects. To make it easy to write API documentation, CommentWeaver provides implicit pointcuts and explicit pointcuts.

7. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming*, pages 144–168. Springer Berlin / Heidelberg, 2005.

[2] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.

[3] W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design With Crosscutting Interfaces. In *IEEE Software, vol.23*, pages 51–60, 2006.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, LNCS 2072*, pages 327–353. Springer, 2001.

[5] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[6] D. E. Knuth. The web system of structured documentation. Technical report, Stanford, CA, USA, 1983.

[7] D. E. Knuth. "Literate programming". *The Computer Journal*, 27(2):97–111, May 1984.

[8] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation: Version 3.0*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[9] S. Kojarski and D. H. Lorenz. Modeling aspect mechanisms: a top-down approach. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 212–221, New York, NY, USA, 2006. ACM.

[10] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003 – Object-Oriented Programming*, pages 219–233. Springer-Verlag, 2003.

[11] H. masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *FOAL 2002 Proceedings of Foundations of Aspect-Oriented languages Workshop at Aspect-oriented software development (AOSD 2002)*, pages 17–26, 2002.

[12] S. Microsystems. Java 2 platform stard edition 5.0 api specification. http://java.sun.com/j2se/1.5.0/docs/api/.

[13] S. Microsystems. Javadoc 5.0 tool. http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/.

[14] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2006. ACM.

[15] A. Organization. The aspectj documentation tool. http://www.eclipse.org/aspectj/doc/next/devguide/ajdoc-ref.html.