Tool support for crosscutting concerns of API documentation

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

Writing detailed API (Application Programming Interface) documentation is a significant task for developing a good class library or framework. However, existing documentation tools such as Javadoc provide only limited support and thus the description written by programmers for API documentation often contains scattering text. Occasionally, it also contains tangling text. This paper presents that this problem is due to crosscutting concerns of API documentation. Then it proposes our new tool named *CommentWeaver*, which provides several mechanisms for modularly describing API documentation of class libraries or frameworks written in Java or AspectJ. It is an extended Javadoc tool and it provides several new tags for controlling how the text manually written by the programmers is scattering and appended to other entries or how it is moved from the original entry to another entry to be tangling. Finally this paper evaluates CommentWeaver by using three class libraries and frameworks: Javassist, the Java standard library, and Eclipse. It showed that CommentWeaver resolves the problems of scattering or tangling text and it adequately reduces the amount of description written by programmers for API documentation.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Aspect-oriented programming, domain-specific language, API documentation

AOSD'10 March 15–19, Rennes and St. Malo, France

1. INTRODUCTION

Writing documents is a significant part of software development [21, 29]. Software developers have to write various documents such as the specifications of the software, bug tracking reports, and users' manual. A number of tools support these documentation activities.

Developers of class libraries and application frameworks have to write API (Application Programming Interface) documentation, which describes classes, methods, and fields in the library or framework. Good libraries or frameworks should have good API documentation, which the users read as a reference manual to learn how to use the software [7]. In Java, the Javadoc tool [24] helps to write API documentation. It enables writing API documentation as comments directly embedded in program source files. These comments are called *doc comments*. Javadoc improves the maintainability of API documentation because developers can easily update the documentation together when they modify a program.

API documentation, however, involves a non-negligible number of crosscutting concerns. These concerns cut across the structure of API documentation, or doc comments. Although modern programming languages such as Java provide several language constructs for modularly describing programming concerns, existing documentation tools such as Javadoc do not provide sufficient support for the modularity. Thus, doc comments often contain scattering or tangling text, which decreases their maintainability. This is also true for the documentation of programs written in an aspect-oriented programming (AOP) language such as AspectJ. AOP languages modularize several crosscutting concerns of programming but not of the documentation. The modularity of doc comments rather gets worse as a programming language provides better constructs for modularization.

To address this problem, this paper proposes our documentation tool named *CommentWeaver*. It is an extended Javadoc tool and provides special tags for modularly describing doc comments for Java or AspectJ programs. When the API documentation of the programs is generated, CommentWeaver makes copies of the doc comments and appends them to the documentation of multiple methods according to the special tags. Thus, the text written by programmers for one method can be automatically appended to the API documentation of other methods related to the original one with respect to the program semantics. For example, a method that will call another method can share the text with the

called method. If a method is advised by an aspect, it can also share the text with that aspect. This eliminates scattering text and improves the modularity of doc comments.

This paper also discusses the applicability of CommentWeaver. We investigate three publicly available class libraries written in Java and Javadoc: Javassist, the standard class library of Java, and Eclipse. We examine how many crosscutting concerns are contained in those doc comments. These concerns can be modularized by CommentWeaver. We also examine how many lines of doc comments are eliminated after we rewrite the original doc comments to be more modular by CommentWeaver. To evaluate the support for aspects, we partly rewrite the Javassist library in AspectJ and apply CommentWeaver to doc comments for aspects.

Our contribution is the following:

- Presenting that API documentation contains crosscutting concerns and existing tools such as Javadoc do not enable modularly describing (*i.e.* implementing) the API documentation.
- Proposing an aspect-oriented simple extension to Javadoc for modular description of API documentation.
- Illustrating its applicability by using three widely used Java class libraries and frameworks: Javassist, the Java standard library, and Eclipse.

In the rest of this paper, Section 2 describes doc comments and problems of existing documentation tools. Section 3 presents CommentWeaver. Section 4 evaluates CommentWeaver by using a few class libraries. Section 5 mentions related work. Section 6 concludes this paper.

2. WRITING DOC COMMENTS

Although writing good API documentation for a library or an application framework is essential to make it really reusable for a wide range of users, current tool support for the documentation is limited. The text of API documentation often involves duplication and thus its source-level representation (*i.e.* doc comments) is scattering or tangling.

2.1 API documentation

Writing API documentation in a program source file has been known as good practice. In the Lisp family of languages, a function definition can include the description of that function. The descriptions in function definitions are collected by a programming tool/environment to be browsable as API documentation. This feature significantly improves developers' productivity when they are writing a program by using a third-party library or application framework. Libraries and application frameworks would be difficult to use without good API documentation.

In Java, the Javadoc tool is widely used for writing API documentation. The descriptions of classes and their members, such as fields and methods, are written as comments surrounded between /** and */. Javadoc collects these comments, which is called *doc comments*, and generates API documentation of the class library or the framework in the HTML format. Integrated development environments such as Eclipse also recognize doc comments. They can show the doc comment of the method selected by a mouse pointer on a code editor, for example. Figure 1 shows three methods, each of which has a doc comment. The @param tag included in the doc comments is a special tag. It specifies that the following text is the name of a method parameter and its description. For example, the @param in the doc comment of the toBytecode method is followed by the description of the out parameter to the method.

Javadoc allows programmers to choose which entities are included in the generated API documentation. Programmers may choose that only public and protected classes and members are included. Normal API documentation describes only those classes and members because the others are invisible from the outside of the library or the framework. This fact is one of the sources of crosscutting doc comments but we discuss this issue later.

2.2 Scattering and tangling

Since Javadoc works as a language processor, doc comments can be regarded as implementation of API documentation. They are source code for generating API documentation. However, their structure is not sufficiently modular. According to our observation, doc comments tend to contain scattering and tangling text. This problem is mainly due to lack of modularization mechanisms for API documentation. Since a modern programming language such as Java provides several constructs for modularization, public classes and methods exposed to the library/framework users do not directly implement all concerns. For example, some concerns are implemented by separate methods invisible from the users. The public methods related to such a concern call the invisible method to achieve separation of concerns. However, the doc comments on that concern cannot be put at this invisible method since the doc comment of the invisible method is not included in the API documentation. The doc comment is redundantly put together with doc comments about other concerns at all the public methods that call the invisible method. This fact causes doc comments to be scattering or tangling. This is another example of crosscutting concerns of aspect orientation [14] or the tyranny of the dominant decomposition [27].

We below illustrate this crosscutting problem by showing a few examples taken from the Javassist library [8]. Javassist is a Java class library for bytecode transformation. It was initially developed by one of the authors and it is currently maintained as open source software of JBoss/Redhat. It has been widely used for a decade by a number of software products including Web application frameworks commercially supported by Redhat. The size of the library (version 3.6) is 53,477 lines of code (LOC) and 9,512 of 53,477 lines are doc comments for Javadoc (18%).

Procedures

We first show an example of doc comments crosscutting across procedure abstractions. A class library often provides multiple methods with the same name but different types of parameters. Since they perform the same function except input parameters, the descriptions of those methods normally have some overlaps.

Figure 1 presents an example of such methods taken from Javassist. The doc comments of the two writeFile methods in the CtClass class share the same text starting with "Once this method is called". This text is also shared with the toBytecode method. Note that the function of these methods, which is converting a class definition into a class file (Java

```
public abstract class CtClass {
        :
 /**
  * Writes a class file represented by this <code>CtClass</code>
  * object in the current directory. Once this method is called,
  * further modifications are not possible any more.
  */
 public void writeFile() throws .. {
     writeFile(".");
 }

 /**
  * Writes a class file represented by this <code>CtClass</code>
  * object on a local disk. Once this method is called, further
  * modifications are not possible any more.
  *
  * @param directoryName it must end without a directory separator.
  */
 public void writeFile(String directoryName) throws .. {
     DataOutputStream out = ...;
     toBytecode(out);
          :
 }

 /**
  * Converts this class to a class file. Once this method is
  * called, further modifications are not possible any more.
  *
  * <p>This method dose not close the output stream in the end.
  *
  * @param out the output stream that a class file is written to.
  */
 public void toBytecode(DataOutputStream out) throws .. {
     throw new CannotCompileException("not a class");
}}
```

Figure 1: Scattering text in the doc comments

bytecode), is implemented by the toBytecode method (of the subclass of CtClass because CtClass is an abstract root class). The two writeFile methods directly or indirectly call the toBytecode method and they are used as helper methods, which construct an appropriate DataOutputStream object before calling toBytecode.

For avoiding the duplication of the text "Once this method...", there should be something like a common doc comment of the three methods and the text "Once this method..." should belong to that doc comment. However, Java does not provide a mechanism for grouping the three methods into a single module or Javadoc does not allow writing a doc comment shared among the three methods. Hence, we must write the doc comments that contain scattering text to the three methods. Note that some duplicated text may not be scattering. For example, if some methods without caller-callee relations share the same text in their doc comments, we do not consider the text is scattering.

The three methods in Figure 1 potentially could be a source of code scattering but they do not contain scattering code because the programmer applied procedural abstraction. The implementation of the core function of the three methods is separated into the toBytecode method and the other two methods call toBytecode for reusing the implementation. However, this separation of a concern by procedural abstraction is not applied to API documentation. The doc comments remain scattering.

The text "Once this method..." must appear in the descriptions of not only the toBytecode method but also the caller methods writeFiles. This is because the users would not read the description of toBytecode when they write a user program that calls writeFile. They would not know

```
public abstract class CtClass {
          :
 /**
  * Defrosts the class so that the class can be modified again.
  *
  * <p>To avoid changes that will be never reflected, the class
  * is frozen to be unmodifiable if it is loaded or written out.
  * This method should be called only in a case that the class
  * will be reloaded or written out later again.
  *
  * <p>If <code>defrost()</code> will be called later, pruning
  * must be disallowed in advance.
  *
  */
 public void defrost() {
    throw new RuntimeException("cannot defrost " + getName());
}}

class CtClassType extends CtClass {
          :

 public void defrost() {
    checkPruned("defrost");
    wasFrozen = false;
}}
```

Figure 2: Tangling text in the doc comment

that writeFiles internally call toBytecode. This fact is an implementation detail that should be hidden from the users according to the information hiding principle [28]. Furthermore, if the toBytecode method were private, both descriptions of the two writeFile methods would have to definitely contain the text "Once this method..." because the description of the toBytecode method would not be included in the API documentation.

Inheritance

Our second example is a doc comment crosscutting along an inheritance hierarchy. A class library or a framework often provides only a public interface (or abstract class) to access some objects internally created. Their actual implementations are given by non-public classes implementing the interface (or subclasses of the abstract class). If these non-public classes show implementation-dependent behavior, which is not mentioned in the specification of that public interface, the API description of the public interface must cover that implementation-dependent behavior.

Figure 2 is another part of the declaration of the CtClass class. This abstract class is used as an interface to objects representing types (or class files). It is extended by several subclasses, which represent primitive types, class types, or array types. A CtClass object is made unmodifiable for avoiding accidental changes after it is converted into a class file. The defrost method in CtClass makes the object modifiable back.

After the first version of Javassist including the defrost method was released, Javassist was updated to have a *pruning* mechanism for reducing memory consumption. However, if this pruning mechanism is on, the defrost method does not work. To indicate this fact, the text "If defrost() will be called..." had to be appended to the description of the defrost method. This is an example of implementation-dependent doc comments. It might be removed if a mechanism with higher compatibility with the defrost method is invented and substituted for the pruning mechanism in future.

A problem in Figure 2 is that the text "If defrost() will

```
public class ClassPool {
     :
 /**
  * Creates a new public class. If there already exists a
  * class/interface with the same name, the new class
  * overwrites that previous class.
  *          :
  * @throws RuntimeException if the existing class is frozen.
  */
  public CtClass makeClass(String name, CtClass superclass)
                throws RuntimeException {
    CtClass clazz = ...;
    return clazz;
 }

 /**
  * Creates a new public interface. If there already exists a
  * class/interface with the same name, the new interface
  * overwrites that previous one.
  *          :
  * @throws RuntimeException if the existing interface is frozen.
  */
  public CtClass makeInterface(String name, CtClass superclass)
                throws RuntimeException {
    CtClass clazz = ...;
    return clazz;
}}
```

```
aspect FrozenChecking {
      :
 before(ClassPool cp, String classname) :
        (execution(* ClassPool.makeClass(String, CtClass))
         || execution(* ClassPool.makeInterface(String, CtClass)))
        && args(classname, ..) && this(cp) {
   CtClass clazz = ... ;
   if (clazz.isFrozen())
      throw new RuntimeException(...);
}}
```

Figure 3: Text that should belong to the doc comment in an aspect

be called..." is about the implementation of CtClassType, a subclass of CtClass, but the text is in the doc comment of CtClass. Since the subclass is not public, the text cannot be attached to the subclass, which is not mentioned in the API documentation. Thus, in the doc comment of CtClass, two documentation concerns are tangling: one is the behavior of defrost in general and the other is the implementation of defrost in the subclass CtClassType.

This tangling decreases the maintainability of the software. Suppose that we invent a mechanism better than the pruning one. We will modify the implementation of defrost in the subclass CtClassType so that Javassist will use our new mechanism. However, we would not notice that we also have to modify the doc comment of the super class CtClass since the source code of CtClassType does not contain any indication of that fact.

Aspects

Our last example is a doc comment of an aspect. Aspect Orientation is a new modularization scheme and aspect-oriented programming languages such as AspectJ [1] provide language constructs for modularizing crosscutting concerns. For example, AspectJ enables scattering implementation code in Java to be grouped and separated into a single module without duplication. This module is called *an aspect*.

Figure 3 presents an example of aspects. This aspect FrozenChecking modularizes the scattering code found in the original code of Javassist. Since several methods in the original ClassPool class (and other classes) confirm that the class is still modifiable before they actually modify the class. The aspect moves all the confirmation code into its before advice.

@quote ((*class-name.*)? *member-name*) (.*export-name*)?

In the doc comment of a method or an advice m,
member-name ∈ { directly called methods from m }

@export (: *export-name*)? { *text* }

text := @quote(..) *text* | ⟨normal text⟩ *text* | ⟨javadoc tags⟩ | ϕ

@weave (*pc*) { *text* }

pc := call (*method-pattern*) | exec (*method-pattern*) |
within (*class-* or *method-pattern*) |
pc && *pc* | *pc* || *pc* | ! *pc*

In the doc comment of an advice, also
pc := JP | JP_CALLER | JP_CALLEE

@liftup { *text* }

Figure 4: Syntax of CommentWeaver tags

Although the aspect improves the maintainability of the confirmation code, it causes scattering text in doc comments. The aspect improves the visibility of when the confirmation code is executed. It also makes the confirmation code removable without modifying the rest of the code when a better mechanism is invented in future. On the other hand, the doc comments of the makeClass and makeInterface methods still contain the text about the confirmation code, which is "@throws RuntimeException if the existing class/interface is frozen.". To modify the text, all the doc comments including this text must be edited. For better modularity of doc comments, this text should be put in the doc comment of the before advice of the FrozenChecking aspect. However, this approach is not acceptable because the aspect is not public and hence the doc comment of the aspect is not included in the API documentation. Even if the aspect were public, the users of makeClass and makeInterface could not notice the note about the confirmation because it is not in the API documentation of these methods. The users would have to see an aspect advises these methods and read the API documentation of the aspect.

3. COMMENTWEAVER

To address the problems mentioned in the previous section, we propose a new documentation system named *CommentWeaver*. It is an extended Javadoc tool and it supports describing API documentation of class libraries and frameworks written in Java or AspectJ.

CommentWeaver allows programmers to modularize crosscutting concerns of API documentation. Javadoc users often write doc comments that contain scattering or tangling text. On the other hand, the users of CommentWeaver can write doc comments in which every concern is described only once at the most appropriate place, for example, the method directly implementing the behavior corresponding to that concern.

When CommentWeaver generates the API documentation from those doc comments, it makes copies of doc comments and appends them to the API documentation of several other methods, which are different from the methods that the doc comments are originally attached to. The appended

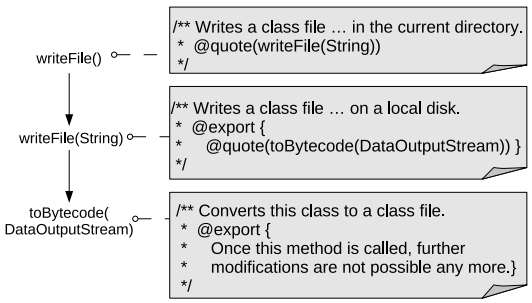

Figure 5: @quote and @export tags in the CtClass class

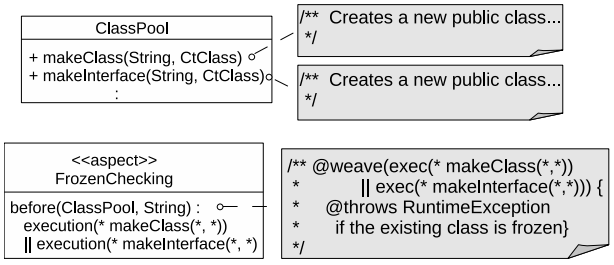

Figure 6: A doc comment moved into an aspect

doc comments are tangled with others and thereby provide comprehensive description of the methods. This generation of the API documentation by CommentWeaver is explicitly controlled according to the special tags written by the programmers. CommentWeaver provides several tags for this as well as the Javadoc tags. The syntax of the CommentWeaver tags is presented in Figure 4.

3.1 Scattering text by procedure abstraction

The crosscutting doc comments caused by procedural abstraction are addressed by the two tags @quote and @export provided by CommentWeaver. These tags are mainly available in the doc comments of methods. The @quote tag is used to refer to the doc comment of another method, which must be called from the method with that @quote tag. When the API documentation is generated, the @quote tag is replaced with the doc comment of the method that the @quote tag refers to. If the doc comment of the method referred to includes the @export tag, only the text following that @export tag is substituted for the @quote tag.

The text is shared only among the methods in the call chain obtained by static analysis. If the text is accidentally equivalent to the text of another method out of a call chain, it is prevented to replace the former text with the @quote tag specifying the latter text. This restriction is for maintainability of doc comments. For example, when a method with @export tag is modified, the text bracketed by @export will be also modified. It will be appropriate that this modification of the text is only propagated along the call chain. In addition, for increasing the maintainability, the argument to the @quote tag must be a method directly called within the method having the @quote tag.

For example, these tags resolve the scattering problem in Figure 1 of Section 2.2. Figure 5 illustrates the result of rewriting the program in Figure 1 with the @quote and @export tags. Note that the text "Once this method..." in the doc comment of the toBytecode method is bracketed by the @export tag. To include the text, the @quote tag is used. See the writeFile(String), which includes it by the @quote tag. The argument to @quote specifies the method to include the doc comment of it. Due to this @quote tag, the duplication of the text is eliminated. The writeFile() method in Figure 5 also has the @quote tag but its argument is the writeFile(String) method. The @quote tag of the writeFile() method is replaced with the text bracketed by @export of the toBytecode method as the @quote tag of the writeFile(String) method is.

In some situations, nevertheless, some developers may want to refer to the text from a method without caller-called relation. For the sake of their need, CommentWeaver provides another tag that is free from the restriction. The detail is mentioned in the later sections.

Multiple @export tags

A doc comment can include multiple @export tags. Since an @export tag can have a name, @quote tags may refer to the names of @exports. For example, as shown below, the doc comment of the toClass method has two @export tags. The doc comment of a caller method, which calls this toClass method could be the following.

```
/**
 * @quote(toClass(CtClass, ClassLoader)).conversion
 * This is only for backward compatibility.
 * @quote(toClass(CtClass, ClassLoader)).warning
 */
 public Class toClass(ClassLoader loader) {
      classPool.toClass(this, loader);
 }

/**
 * @export : conversion {
 *   Converts the class to a <code>java.lang.Class</code> object.
 * }
 *
 * Do not override this method any more at a subclass because
 * <code>toClass(CtClass)</code> never calls this method.
 *
 * @export : warning {
 *   <p><b>Warning:</b> A Class object returned by this method
 *   may not work with a security manager or a signed jar file
 *   because a protection domain is not specified.
 * }
 *   :
 */
 public Class toClass(CtClass ct, ClassLoader loader) { ... }
```

In this doc comment, the text "This is only..." is substituted for the second sentence "Do not override..." of the doc comment of the called method. The rest of the doc comment is the same.

3.2 Scattering text by aspect

Although aspects modularize crosscutting concerns for programming, they do not for doc comments as we mentioned in Section 2.2. Scattering text is still included in multiple doc comments. Furthermore, these doc comments are of the target methods advised by the aspect. They should be attached to the aspect directly implementing the behavior described by that text.

The crosscutting doc comments caused by aspects is addressed by the @weave tag. It is available in the doc comments of AspectJ's advices. It is used to append the following text to methods selected by the argument. To select methods, the argument to @weave is the pointcut, such

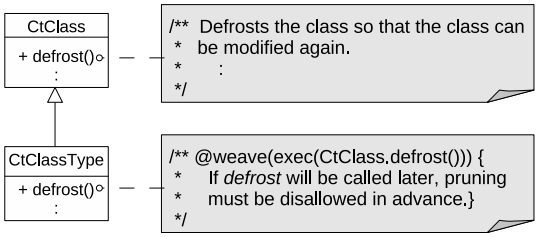

Figure 7: A doc comment moved to a public super class

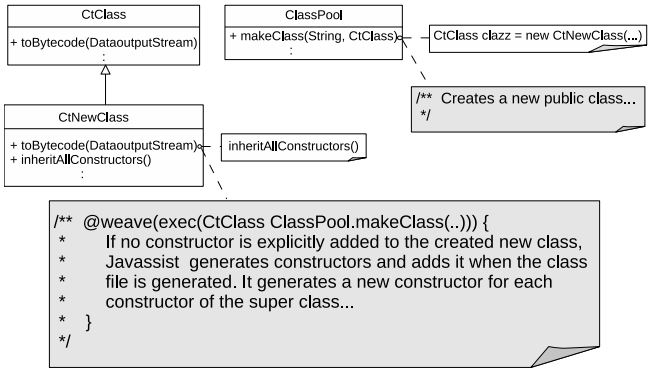

Figure 8: A doc comment moved to a factory method

as call and exec, which are borrowed from AspectJ. While @quote pulls the text from another method, @weave pushes the text to another. We illustrate the use of @weave by rewriting the program that we presented in Figure 3. Figure 6 shows the result of the rewrite for CommentWeaver. The difference between Figure 3 and 6 is that the text starting with @throws is moved from the two methods makeClass and makeInterface into the aspect and is bracketed by @weave. In Figure 6, the text is included in the doc comment of the code block directly implementing the behavior described by that text. Duplication of the text is now eliminated.

As shown in Figure 6, developers may have to enumerate method names as the arguments of @weave. To avoid the repetition of the description of AspectJ pointcut, CommentWeaver provides the special variables JP, JP_CALLER, and JP_CALLEE. For example, the @weave tag in Figure 6 can be simplified as the following:

```
@weave(JP) { ... }
```

Since CommentWeaver is a compile-time tool, the variable JP represents join point shadow [22] to determine the methods that doc comments of an advice is appended to. The @weave with JP can append the doc comment to the methods containing the join point shadow selected by the pointcut of that advice body. In Figure 6, the join point shadow is the makeClass and makeInterface methods, which are selected by the two execution pointcuts. Since the JP uses join point shadow, only the so-called accessor pointcuts call, execution, set, and get are considered. cflow and if pointcuts are ignored.

Developers might think the description of RuntimeException should be included also in the API documentation of the caller methods that call the makeClass and makeInterface in the ClassPool class. If so, the doc comments of the before advice could be modified into the following.

```
@weave(JP || JP_CALLER) { ... }
```

The variable JP_CALLER represents the caller methods.

3.3 Tangling text by inheritance

The @weave tag is also available in the doc comment of a method. For better modularity, it enables separating doc comments which would be otherwise crosscutting an inheritance hierarchy. The text bracketed by this tag can be appended to the API documentation of the overridden method in the super class or an implemented interface.

For example, Figure 7 is the result of rewriting the program in Figure 2 with the @weave tag. The text "If defrost() will be..." is moved from the CtClass class to the subclass CtClassType. On the other hand, since the text is bracketed by the @weave tag, when the API documentation is generated, it is appended by CommentWeaver to the API documentation of the defrost method in the CtClass class. This rewriting improves the separation of concerns. The description about implementation-dependent behavior is attached to the method directly implementing that behavior although the method is not visible to the library/framework users.

For the tangling text by inheritance, since the target specified by the argument to @weave is apparent, CommentWeaver provides the @liftup tag that takes no argument. Furthermore, while @weave expects developers to specify which super class the target is defined in, @liftup itself tries to find the target recursively through the hierarchy. The description in Figure 7 can be replaced with "@liftup { If defrost will be ... }".

3.4 Another example: weaving text at an appropriate location

The example in Section 3.3 showed that a non-public subclass causes tangling text in its public super class and CommentWeaver can address this problem. A non-public subclass also causes tangling text in a class declaring a factory method for the non-public class.

The CtClass of Javassist is a public abstract class and a variable of the CtClass type always refers to an instance of its concrete subclass such as CtClassType and CtNewClass. CtNewClass is another non-public subclass and thus invisible from the users' viewpoint. It is instantiated by a factory method makeClass in the ClassPool class, which is public. The toBytecode method declared in CtNewClass overrides its super's method and it implements the common behavior of toBytecode described in the doc comment in the super class CtClass. Although the description of the doc comment in the super class was sufficiently general, a Javassist user queried detailed behavior of toBytecode (implemented in CtNewClass) [3]. Thus, the Javassist developers decided to add extra text to the API documentation and they chose as an appropriate place the factory method declared in ClassPool since only that factory method returns an instance of CtNewClass. Other factory methods return instances of the other

subclasses of CtClass. The added text is not applicable to instances of the other subclasses.

This is another example of tangling text but CommentWeaver can address this problem. As shown in Figure 8, the @weave tag enables us to include the extra text in the doc comment of the toBytecode method in CtNewClass. The text following @weave is copied to the factory method makeClass in ClassPool from the toBytecode method, which implements the behavior described by that text.

3.5 Semantics

We show the semantics of the @quote, @weave, and @liftup. To simplify the presentation, the @export is not taken into consideration. Let a method m be the following form:

$$m = /** \ s_1..s_h q_{m_1}..q_{m_i} w_1..w_j l_1..l_k \ */ T_1 \ \mu(T_2 \ x)\{e\}$$

where s is the normal text or javadoc tag, q_m represents that an @quote tag specifying a method m for its parameter, w is an @weave tag, and l is a @liftup tag. For m, we define helper functions id and doc; $id(m) = \mu$, and $doc(m) = s_1..s_h q_{m_1}..q_{m_i} w_1..w_j l_1..l_k$. We then define helper functions for w. Suppose that w_j is the following:

$$w_j = \text{@weave}(pc) \ \{ \ s_1^{(j)}..s_u^{(j)} q_{m_1^{(j)}}..q_{m_v^{(j)}} \ \}$$

where the bracketed text consists of the normal text and javadoc tags (represented by $s_u^{(j)}$), and @quote tags such as $q_{m_v^{(j)}}$, which takes $m_v^{(j)}$ for its parameter. Two functions are defined: $pce(w_j) = pc$, and $wbody(w_j) = s_1^{(j)}..s_u^{(j)} q_{m_1^{(j)}}..q_{m_v^{(j)}}$. Similarly, let l_k be

$$l_k = \text{@liftup} \ \{ \ s_1'^{(k)}..s_x'^{(k)} q_{m_1'^{(k)}}..q_{m_y'^{(k)}} \ \}$$

and define a helper function to get the bracketed text: $lbody(l_k) = s_1'^{(k)}..s_x'^{(k)} q_{m_1'^{(k)}}..q_{m_y'^{(k)}}$.

We next show the semantics of generating the API documentation. Generating the API documentation of m is to compute:

$$[\![doc(m)]\!]_{m,\Sigma} + advices(m)$$

The operator $[\![-]\!]_{m,\Sigma}$ expands the tags in the given text to generate the API documentation for a method m. Σ is a set of methods. Its initial value is an empty set. First, since s is the text including no tags, $[\![s]\!]_{m,\Sigma} \rightarrow s$. Thus, $[\![-]\!]_{m,\Sigma}$ is distributive.

$$\begin{aligned}
&[\![s_1..s_h q_{m_1}..q_{m_i} w_1..w_j l_1..l_k]\!]_{m,\Sigma} \\
&\rightarrow s_1..s_h [\![q_{m_1}..q_{m_i} w_1..w_j l_1..l_k]\!]_{m,\Sigma} \\
&\rightarrow s_1..s_h [\![q_{m_1}]\!]_{m,\Sigma} .. [\![q_{m_i}]\!]_{m,\Sigma} [\![w_1]\!]_{m,\Sigma} .. [\![w_j]\!]_{m,\Sigma} [\![l_1]\!]_{m,\Sigma} .. [\![l_k]\!]_{m,\Sigma}
\end{aligned}$$

$[\![w]\!]$ and $[\![l]\!]$ are evaluated as follows:

$$\begin{aligned}
&[\![w]\!]_{m,\Sigma} \rightarrow [\![wbody(w)]\!]_{m,\Sigma} \\
&[\![l]\!]_{m,\Sigma} \rightarrow [\![lbody(l)]\!]_{m,\Sigma}
\end{aligned}$$

The rules above means that CommentWeaver first evaluates @quote tags and then @weave and @liftup tags. Suppose that @weave (or @liftup) appends the text to a method m. This text is not quoted by @quote from the method m to another method.

The evaluation rule for $[\![q]\!]_{m,\Sigma}$ is this:

$$\frac{\begin{array}{c} m \text{ calls } m' \\ doc(m') = s_1..s_h q_{m_1}..q_{m_i} \\ m \notin \Sigma \end{array}}{[\![q_{m'}]\!]_{m,\Sigma} \rightarrow s_1..s_h [\![q_{m_1}]\!]_{m',\Sigma\cup\{m\}} \ .. \ [\![q_{m_i}]\!]_{m',\Sigma\cup\{m\}}}$$

where the first line represents that the method m and m' are in the same call chain, that is, m calls statically m' in its method body. To avoid recursively expanding @quote, a history of the expansion is recorded in Σ. If m is not in Σ, $q_{m'}$ is reduced to the doc comment of m'. Note that m is added to Σ after that.

If m is already in Σ, $[\![q_{m'}]\!]_{m,\Sigma}$ is deleted as shown below. ϕ represents empty.

$$\frac{\begin{array}{c} m \text{ calls } m' \\ m \in \Sigma \end{array}}{[\![q_{m'}]\!]_{m,\Sigma} \rightarrow \phi}$$

If m does not call m', then a compile error is reported.

The function *advices* collects the text appended by @weave and @liftup.

$$\frac{[\![w_i]\!]_{n_j,\phi} \in woven(m) \text{ for } i \in 1..a, j \in 1..b}{advices(m) = \sum_{i,j} [\![w_i]\!]_{n_j,\phi}}$$

Here, $woven(w)$ is a helper function. It receives a method m and returns a set of $[\![w]\!]$, where the pointcut of w matches the given m.

$$\begin{aligned}
woven(m) = \{[\![w]\!]_{n,\phi} \mid \exists n : method, \ w \in doc(n), \\
pce(w) \ matches \ m\}
\end{aligned}$$

The pointcut *pce(w)* matches a method m if the pointcut selects a joinpoint included in m. For example, if *pce(w)* is exec(m), it surely matches m. If *pce(w)* is call(m') and m calls m', it will match m. The *pce(w)* may be liftup(m) because @liftup is transformed into @weave(liftup(m)), where liftup is a pointcut only internally available to select the methods in super classes with the same signature as m. The liftup(m) matches a method m' if m' is one of the methods in the super classes.

4. CASE STUDIES

As shown in the previous sections, CommentWeaver improves the modularity of the description for API documentation. For example, we have already presented that the examples shown in Section 2.2 can be rewritten to be more modular by CommentWeaver. This section discusses the applicability of CommentWeaver to existing class libraries.

4.1 Javassist

We first investigated how much scattering or tangling text appears in the doc comments of the Javassist bytecode transformation library. We counted the number of doc comments including such text by using a software tool we developed for finding scattering text in Java source files. We investigated 3 packages among 12 public ones of Javassist 3.6. We selected the packages containing more than 10 classes or interfaces: javassist, javassist.bytecode, and javassist.bytecode.annotataion packages.

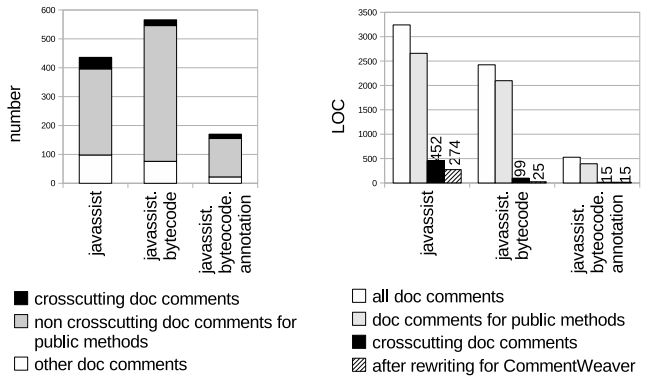

Figure 9: The doc comments for Javassist

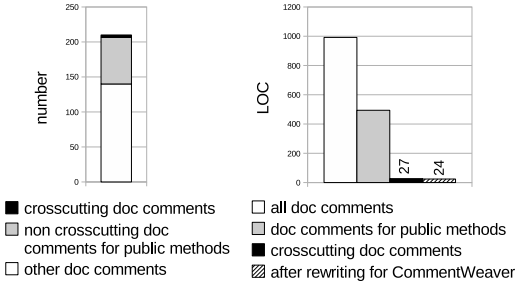

Figure 11: The doc comments for Eclipse (on average per package)

Figure 9 illustrates the result of our investigation. It shows that a fair number of doc comments include scattering or tangling text[1]. The left chart in the figure presents the number of the doc comments and the right chart presents the lines of code (LOC) of the doc comments. The javassist package contains 436 doc comments in total and 338 doc comments are for public methods (the others are for classes and other entities). We found that 40 of 338 doc comments for public methods are crosscutting and hence they contain scattering or tangling text. The ratio is 12% (17% in LOC). For the javassist.bytecode package, 4% of the doc comments (5% in LOC) are crosscutting and, for the javassist.bytecode.annotation package, 10% of the doc comments (4% in LOC) are crosscutting. The results reveal that CommentWeaver contributes to improve the modularization of about one-tenth of the doc comments.

We then investigated how many lines of doc comments can be reduced by using the @quote and @export tags of CommentWeaver. The right chart in Figure 9 presents the result. For the javassist package, the crosscutting doc comments were reduced from 452 to 274 LOC. Thereby, the doc comments for public methods were reduced from 2659 to 2481 LOC (7% reduction). For the javassist.bytecode package, the doc comments for public methods were reduced from 2097 to 2023 LOC (4% reduction). However, for the javassist.bytecode.annotation package, the doc comments for public methods were not reduced at all (0% reduction). This is because the size of all the scattering text found in the crosscutting doc comments is only one line. We substituted a @quote tag for such one-line text but the @quote tag also occupies one line. The total number of lines did not change.

We finally present the number of doc comments including tangling text. We found five doc comments included tangling text (73 LOC). This number indicates how frequently the @liftup tag is needed. This tag does not contribute to the reduction of doc comments but it improves the maintainability of them. All the doc comments we found were for the methods of the CtClass class in the javassist package. The other packages did not contain such doc comments. Note that the CtClass class is the only public class that has non-public subclasses in the three packages. Since the doc comments for the public methods of the CtClass class are 542 LOC, 13% of these doc comments require the @liftup tag. We lists the details of these doc comments in Table 1.

[1]If two doc comments share the same scattering text, we counted one as a doc comment including scattering text. We did not count the other.

4.2 The standard library of Java 6

As a larger class library, we also investigated the standard class library of the Java Platform, Standard Edition 6 (Java 6). We selected only the packages that contain more than 100 public methods and more than 1000 LOC of doc comments for the public methods.

Figure 10 illustrates the result. This shows the number of crosscutting doc comments, which contain scattering or tangling text, in each package. On average, 20% of the doc comments for public methods are crosscutting ones. All the crosscutting concerns contained only scattering text. They did not contain doc comments that contain tangling text and thus we could not use @liftup for improving the maintainability.

Figure 10 also presents the size of the crosscutting doc comments. It also presents the size of these doc comments after we rewrote them by using @quote and @export tags of CommentWeaver. After the rewrite, the size was reduced by 3% on average.

4.3 Eclipse

We finally investigated the Eclipse Platform (Release 3.3). Since Eclipse is a framework hosting various development tools implemented as a plugin, the API documentation is a significant part of the products. The plugin developers read this documentation to understand how to connect their plugins to the platform.

Eclipse consists of 204 packages. As Figure 11 presents, on average, each package has 140 doc comments (992 LOC). Among them, 67 doc comments (494 LOC) are for public methods in the package. They included 3 crosscutting doc comments (27 LOC) per package. Thus, 4% of the doc comments for public methods were crosscutting ones.

Almost all the crosscutting doc comments contained scattering text. Hence, for most doc comments, the @quote and @export tags of CommentWeaver were applicable. After we rewrote the doc comments by using those tags, the size of the doc comments was reduced from 27 to 24 LOC on average (10% reduction).

The crosscutting doc comments that the @liftup tag was applicable to were not zero . In total, we found 107 crosscutting doc comments that contained tangling text. The @liftup tag contributes to the API documentation of Eclipse.

method name	behavior	a note about the current implementation mentioned in the tangling text
prune	discards unnecessary attributes	a performance note
defrost	defrosts the class so that it can be modified again	a conflict with another function
makeNestedClass	makes a new public nested class	a functional limitation
getModifiers	returns the modifiers for the class	clarifying ambiguity
getClassFile2	returns a class file for this class	inconsistency with the specification

Table 1: The tangling text in the doc comments for the CtClass class

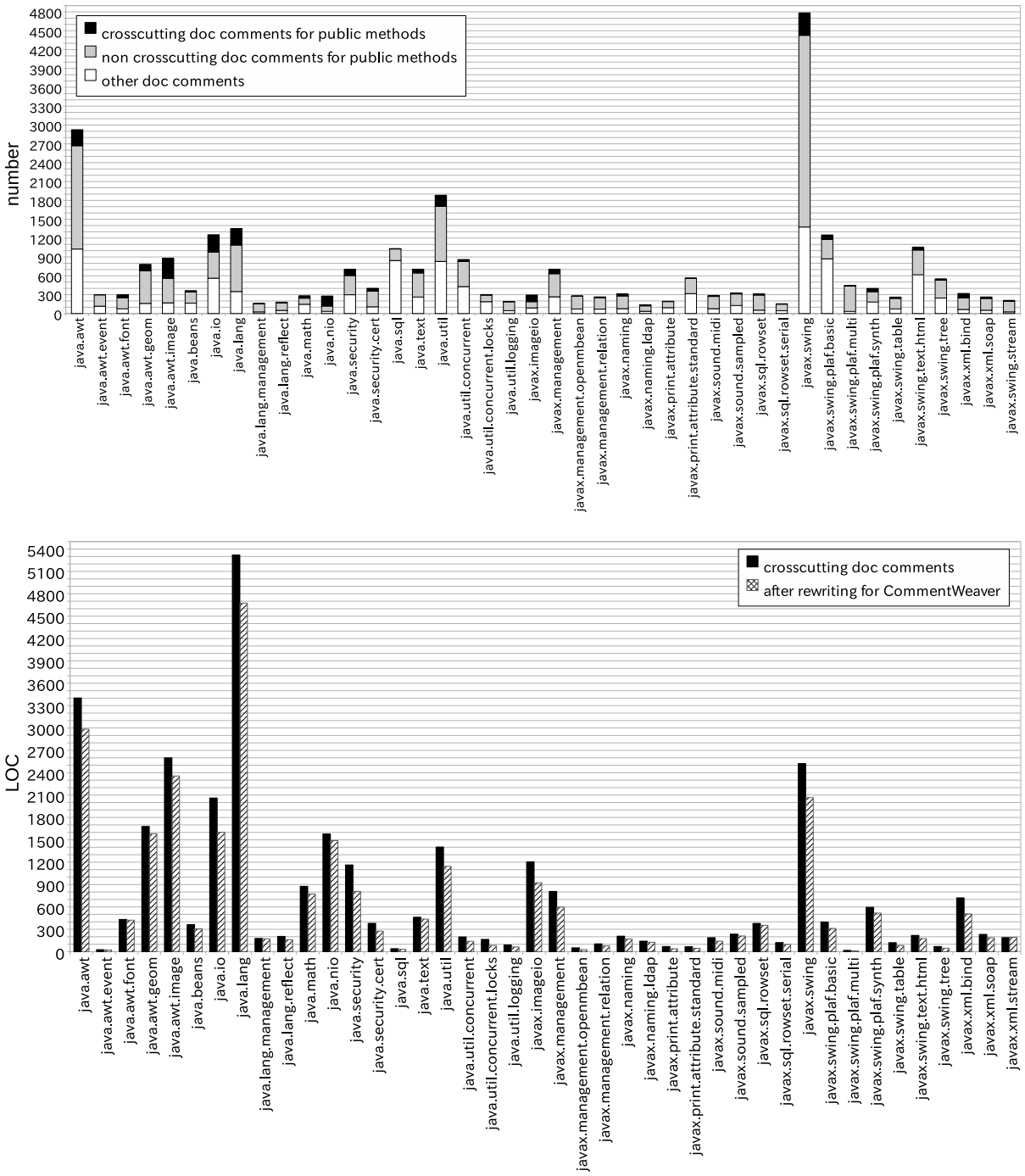

Figure 10: The crosscutting doc comments in Java 6

aspect name	LOC	# of advices (# of advised methods)	needs doc comments	original (LOC)	AspectJ (LOC)	call && within
CtClassCaching	308	16 (18)		0	0	2
FrozenChekcing	111	3 (5)	Yes	3	1	2
ModifyChecking	206	14 (58)	Yes	22	8	2
CodeAttributeCopy	57	2 (2)	Yes	3	3	
ExistingTest	74	1 (1)	Yes	2	2	
InsertionHandling	20	1 (2)	Yes	2	1	
NotFoundExceptionHandling	32	2 (2)	Yes	2	2	
ProxyFactorySynchronization	13	1 (1)		0	0	1

Table 2: The aspects implemented for Javassist

4.4 An AspectJ version of Javassist

CommentWeaver provides support for writing doc comments for aspects. To investigate this support, we partly rewrote Javassist in AspectJ. During this rewrite, we implemented eight aspects:

- CtClassCaching:
  caches class objects
- FrozenChecking:
  checks if the object is frozen
- ModifyChecking:
  checks if the object has been already modified
- CodeAttributeCopy, InsertionHandling, NotFoundExceptionHandling :
  catches a thrown exception, and then throws a different exception
- ExistingTest:
  checks if the member object is duplicated
- ProxyFactorySynchronization:
  manages synchronization

We also wrote doc comments for these aspects with CommentWeaver. Table 2 lists details of the aspects. The column "LOC" indicates the number of lines of the aspect. The column "# of advices" indicates the number of the advice bodies contained in the aspect. The column "# of advised methods" indicates the number of the methods advised by the aspect.

The column "needs doc comments" indicates whether or not the doc comments of the aspect must be appended to the API documentation of the advised methods. The number of "Yes" represents the usefulness of the mechanism of CommentWeaver for automatically copying doc comments from aspects to classes. As Table 2 presents, if an aspect implements a functional concern, then that concern must be described in the API documentation of the advised classes. CommentWeaver is useful for writing doc comments for that concern. On the other hand, if an aspect implements a non-functional concern, then doc comments are unnecessary for that aspect.

The column "original" indicates the size of the doc comments in the original Java version. These doc comments describe concerns that were separated into aspects after the program was rewritten in AspectJ. The column "AspectJ" indicates the size of the doc comments for the aspect. For example, the ModifyChecking aspect modularized not only scattering code for checking but also scattering text for doc comments (22 LOC) into one module (8 LOC). Since duplicated text is eliminated, the size of the doc comments was reduced in the AspectJ version. Some aspects were heterogeneous and hence the doc comments did not contain duplicated text. The size of the doc comments did not change between Java and AspectJ. In total, the size of the doc comments was reduced by 50% after the program was rewritten in AspectJ with CommentWeaver.

The column "call && within" indicates the number of the advice bodies with the call and withincode pointcuts. The numbers at this column show the number of the doc comments that require the variable JP_CALLEE to append the description to the callee-side method as well as the caller-side method. Table 2 shows that the JP_CALLEE was necessary for several cases.

5. RELATED WORK

Javadoc and Ajdoc.

The work most related to CommentWeaver is Javadoc and Ajdoc. We have already compared Javadoc and CommentWeaver in Section 2.2 and mentioned that Javadoc has a problem of crosscutting concerns. Some people might say that similar functionality can be achieved by the @see tag. The @see tag adds hyperlinks to the API documentation of a method, which library users can click to read the related documentation. However, the @see tag does not make every entry of the API documentation self-contained. The @see tag does not clearly indicate whether or not it refers to an important note the users must read. Furthermore, asking the users to follow hyperlinks is not user-friendly. According to our experiences, library users are not always careful to click @see before using a library method [2].

Ajdoc [26] is another Javadoc-like tool for AspectJ. It enables us to attach a doc comment to an advice body in an aspect. However, unlike CommentWeaver, Ajdoc includes a doc comment only in the API documentation of that advice. It is never appended to the API documentation of the

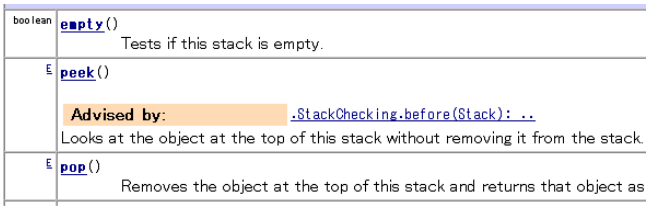

Figure 12: A hyperlink in the API documentation generated by Ajdoc

methods where the advice is woven. Instead, Ajdoc appends hyperlinks to the API documentation of the method, which library users can click to read the API documentation of the advice. For example, Figure 12 shows part of the API documentation of the java.util.Stack class refactored in AspectJ. The entry of the peek method has a hyperlink to a before advice of the StackChecking aspect, which advises the peek method. This hyperlink is another solution of the problem mentioned in Section 2.2 but it exposes a detail of the implementation of the class library, that is, that the Stack class is implemented by using an aspect. This fact is not an interest of the library users and thus it should be hidden.

Another problem of Ajdoc is that it does not provide a mechanism corresponding to the @weave tag of CommentWeaver. For example, the pop method in the Stack class in Figure 12 calls the peek method. Hence, the behavior added to peek by the StackChecking aspect is also added to pop indirectly. However, the API documentation of pop does not provide any clues to know this fact. The entry of the pop method does not have a hyperlink or directly contain the doc comment of the aspect.

The documentation of the CLOS Metaobject Protocol.

Kiczales *et.al* reported the significance and difficulty in writing good API documentation of class libraries according to their experiences of the design and implementation of the CLOS Metaobject Protocol [15]. Since a class library is often extensible by subclassing, the documentation must mention the internal structure of the library and hence writing the documentation is complex. The modularity and maintainability is significant and advanced tool support like one by CommentWeaver is requisite.

Verifying the specifications.

A good library/framework must have good documentation to avoid incorrect use. However, good documentation is not a silver bullet. It is also important to verify a program correctly uses the library/framework. We can see this approach, for example in the design by contract [23, 20], the typestate checking [10, 6], and the FUSION analysis [13]. The FUSION analysis is useful for specifying framework constraints such as the semantics constraints between multiple objects.

Literate programming.

WEB [17], CWEB [19], and FWEB [5] are languages based on the concept of *literate programming* [18], which promotes better documentation of programs. WEB consists of two languages, TEX for writing documentation and Pascal for programming. In WEB, source files contain a Pascal program and the TEX text for improving the readability of that Pascal program. The *WEAVE* operation generates a well-formatted document describing the program. WEB is one of the early systems that promote programmers to write a program and its documentation in the same file.

Approaches to understand crosscutting structures.

Understanding crosscutting structures in a program is not a simple task. This is also true when a program is written in an aspect-oriented programming language such as AspectJ because of their obliviousness property [11]. CommentWeaver addresses this problem by encouraging programmers to write good documentation in a modular way parallel to the program structure. On the other hand, there have been also several language constructs proposed so far to address this problem.

For example, the Aspect-Aware Interface (AAI) [16] is a new kind of interface for addressing this problem. The API documentation generated by CommentWeaver can be regarded as concrete presentation of AAI since the documentation of methods directly mentions the influence of the aspects advising those methods. A difference between AAI and CommentWeaver is that AAI is a language construct or a conceptual model while CommentWeaver is a documentation tool.

Open Modules [4, 25] and XPIs (crosscutting programming interface) [12] are language constructs for addressing the obliviousness property. Their idea is to let programmers declare module interfaces for pointcuts. The programmers must explicitly specify selectable join points from external clients. These interfaces for pointcuts help programmers take care of the selectable join points when they modify the implementation of the module. The approach of Open Modules and XPIs is to restrict possible crosscutting structures whereas the approach of CommentWeaver is to improve the readability of the documentation so that they can avoid problems due to the obliviousness property.

Active models [9] represent crosscutting structures of an AspectJ program. ActiveAspect, which is the tool based on the active models, presents a node-and-link diagram for representing an interesting slice of the crosscutting structure in an AspectJ program. This tool helps programmers understand the crosscutting structure of a program but the presentation by ActiveAspect is different from the presentation by CommentWeaver.

6. CONCLUDING REMARKS

This paper presents our new documentation tool named *CommentWeaver*. It provides a mechanism for modularly describing API documentation, which includes a fair number of crosscutting concerns. According to our experiments using three publicly-available class libraries, which are Javassist, the standard Java library, and Eclipse, 4 to 20% of doc comments written for Javadoc were crosscutting ones. CommentWeaver contributed to the modularity of those crosscutting doc comments. In fact, the size of those doc comments was reduced by up to 10% after the rewrite for CommentWeaver. CommentWeaver is also useful for programs written in AspectJ.

7. REFERENCES

[1] Aspectj project. http://www.eclipse.org.aspectj/.

[2] [#JASSIST-102] Bug report for CtNewMethod.copy - JBoss issue Tracker. https://jira.jboss.org/jira/browse/JASSIST-102.

[3] [#JASSIST-68] Remove limitation on public constructors - jboss.org JIRA. https://jira.jboss.org/jira/browse/JASSIST-68.

[4] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *ECOOP '05 : Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 144–168. Springer Berlin / Heidelberg, 2005.

[5] A. Avenarius and S. Oppermann. Fweb: a literate programming system for fortran8x. *SIGPLAN Not.*,

25(1):52–58, 1990.

[6] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 301–320, New York, NY, USA, 2007. ACM.

[7] Joshua Bloch. How to design a good API and why it matters. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507, New York, NY, USA, 2006. ACM.

[8] Shigeru Chiba. Load-time structural reflection in Java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag.

[9] Wesley Coelho and Gail C. Murphy. Presenting crosscutting structure with active models. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software developmen t*, pages 158–168, New York, NY, USA, 2006. ACM.

[10] Robert DeLine and Manuel Fahndrich. Typestates for objects. In *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Berlin / Heidelberg, 2004.

[11] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.

[12] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Y uanfang Cai, and Hridesh Rajan. Modular Software Design With Crosscutting Interfaces. In *IEEE Software, vol.23*, pages 51–60, 2006.

[13] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. G riswold. An overview of AspectJ. In *ECOOP '01 - Object-Oriented Programming: 15th European Conference, LNCS 2072*, pages 327–353. Springer, 2001.

[15] Gregor Kiczales and John Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 435–451, New York, NY, USA, 1992. ACM.

[16] Gregor Kiczales and Mira 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.

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

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

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

[20] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML, 2003.

[21] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[22] Hidehiko Masuhara, Gregor Kiczales, and Chris 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.

[23] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

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

[25] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh 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.

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

[27] Harold Ossher and Peri Tarr. Multi-demensional separation of concerns in hyperspace. In *Position paper at the ECOOP'99 Workshop on Aspect-Oriented Programming*, June 1999.

[28] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[29] David Lorge Parnas. Document based rational software development. *Know.-Based Syst.*, 22(3):132–141, 2009.