A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

Tool support for Modularized Documentation at the Design and Implementation Phase
実装時ドキュメントへのモジュール構造の導入と支援ツールの提案

Michihiro HORIE

Dissertation Chair:
Shigeru CHIBA

Abstract

Through software development, writing documents is a significant task although the main pillar of software development is programming. Software developers have to write various documents such as the specifications of the software, bug tracking reports, and users' manual. During implementations, writing documents in program source files 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 documentation views.

While modern programming languages provide several language constructs for modularly describing programming concerns, there is no sufficient support for the modularity in documentation. Even if programs are well modularized, documentation is not necessarily modularized. Thus, documents often contain scattering or tangling text, which decreases their maintainability and reusability.

This thesis proposes tool supports for modularized documentation in program source files. During implementation, documentation helps developers to understand a behavior of an encapsulated module such as a method and class. Our tool named *AspectScope* automatically generates woven documentation and shows it via an IDE. When software is released, the precise API documentation is required because insufficient documentation will mislead user programmers. Our documentation tool named *CommentWeaver* enables to generate precise documentation that are woven from modularized documents written in source codes. Finally, after software is released, it often evolves through several versions. Sometimes new specifications will be added, or sometimes simple refactorings will be given. Our tool named *Universal AOP* enables to write robust documentation against refactorings.

These three tools are developed on the basis of the notion of the aspect orientation. They support to describe documents for programs written in Java and AspectJ.

Acknowledgements

This thesis was supervised by Shigeru Chiba. I would like to express my deep gratitude to him. I also profoundly thank Awais Rashid, who supported me for our collaborating research at Lancaster University in the United Kingdom. Ruzanna Chitchyan and Phil Greenwood gave me a plenty of constructive comments and criticism with spending a lot of time. Steffen Zschaler gave me a chance to reconsider what my thesis is essentially tackling on. Joost Noppen and Safoora Shakil Khan taught me to have a positive thinking to write my thesis. Yoshisato Yanagisawa always gave me useful advices and supported me to advance my research. The committees are organized by Shigeru Chiba, Masataka Sassa, Satoshi Matsuoka, Ken Wakita, and Kazuyuki Shudo.

I thank all of my colleagues in the research group in Tokyo Institute of Technology. Especially, I would be deeply grateful to Takeshi Azumi, who came to encourage me during my stay in Lancaster. Also, Takeshi Azumi, Kohei Betchaku, Sungho Arai, Kei Kanazawa, and Yuki Teramoto significantly helped me to conduct time-consuming case studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing documents is a significant part of software development [60, 74]. Software developers have to write various documents such as the requirements, specifications, bug tracking reports, and users' manual. Documentation of the requirements and specification phase is imperative, because concrete programs have not been implemented yet, and so, documentation is the only tool to define the whole behavior of software. On the other hand, documentation at the design and implementation phase (and later phases such as the maintenance phase) must not be written necessarily because the main pillar of these phases is programming. However, even during implementation, documentation is still very significant to make other developers understand the behavior of a program before investigating its implementation. It is also important to make developers themselves remind the behavior of a program. However, unlike in the requirement and specification phase, documentation in the design and implementation phase has to be paid attention to the language constructs given by programming languages such as methods. There may be many similar documentation in one program because of the methods that provide a similar functionality. Besides, new programming paradigms have been developed such as Object-Oriented Programming (OOP), and recently Aspect-Oriented Programming (AOP). New programming languages for them have been also released, and some of them are widely used among academia and industries. Therefore, a new documentation system is needed that can be applied to these different language paradigms and its languages.

However, techniques to write documentation have not been so much changed since the Lisp family of languages had been developed: documentation has no modular structure, and each documentation is just written on/in its function definitions. Despite this fact, writing documentation is unexpectedly burdensome throughout software developments. This is because the quality and quantity of documentation is required to develop good software. For example, the low-quality documentation will mislead user programmers who write their software by reading the documentation. Also, less documentation will result in careful investigation of its implementations in the end. Both of these situations mean that insufficient documentation will become worthless after all.

To address these requirements of writing documentation with high quality and large quantity at the design and implementation phase, we consider that modularization mechanisms should be introduced into documentation. Separating each documentation into modules should increase the maintainability and reusability as well as the modularity in programming languages bring these benefits to developers. We also consider that description of documentation should be appended on each programming module as *comments* because comment-style description is easy to apply to new language constructs. In addition, describing with the natural language is intuitive to developers, so, we did not think about any special constructs to define specifications in comments.

1.1 Motivating Problem

There are several techniques and tools to write documentation at the design and implementation phase. However, none of them addresses our motivating problem, that is, modularization of documentation. Although writing documentation has been considered as good practice, documentation has become more important in OOP and AOP. One of the reason is that the encapsulation of programs by classes hides implementation details, and thus, it becomes difficult to understand what is implemented in an encapsulated module. This notion is apparently paradoxical: OOP had been developed for better reusability and maintainability of programs. Although the maintainability increases by separating concerns into many modules, call hierarchies of methods in OOP become deeper in practice [17]. This fact tells that methods in a long call chain need to be explained what they encapsulate. Therefore, documentation and long names for each method are needed to make developers guess the encapsulated detail. Another benefit in OOP is reusability: a module can be replaced with newly defined module in a program without touching the other source codes. For example, framework users can insert new modules into the framework to change its behavior. However, to exert this benefit, developers have to know precisely how to replace a module. A user-defined class might have to extend a certain super class and override some methods.

Thus, documentation is not the main tool to develop a program but very useful to understand existing modules. Especially, since AOP makes program understanding more difficult than OOP, documentation is youthful to do modular reasoning. The only thing developers have to do is to write documentation before/after implementing a module whether or not the module will be declared as public.

When class libraries and application frameworks are released, developers of them 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 [15]. In Java, the Javadoc tool [67] 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.

Literate programming is an approach by Knuth to write program snippets with documentation. In this programming style, documentation is the main construct in source codes rather than programs. This system is developed for functional languages such as Pascal. WEB, which is the language based on the notion of literate programming, 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. However, the problem is that writing documentation with OOP or other paradigms is open issues. Concretely, there is no specific notion in literate programming to construct class component with documentation. Creating inheritance relation is also not mentioned. Besides, to realize these object components only with documentation seems to be difficult.

1.2 Approach by this thesis

To enable modular description of documentation at the implementation and design phase, we propose documentation tools for each purpose: *AspectScope* for viewing modularized document through Eclipse IDE, *CommentWeaver* for generating API documentation, and Universal AOP for obtaining less fragile documentation through refactorings. Our tools can be used along with procedural languages, Object Oriented Programming languages, and even Aspect Oriented programming languages. By using our tools, documents written in the natural languages can be separated into each module along the programming modules. To compose modularized documents, our tools basically need *hints* that users have to give. As the clues, we provide several kinds of tags that can be put within documents. Since modularized documents can be not only append just before/after other documents but also inserted into others, we will call this process as *weaving* of documentation from this chapters.

Our three tools are developed for two criteria: the amount of tagging and the accuracy of woven documentation. When accurate documentation is needed, developers have to put lots of tagging to correctly weave documents. On the other hand, when developers do not want to put so much tagging, the accuracy of woven documentation will become lower. It depends on each development phase whether developers always need lots of tagging or not. When developers are implementing programs, the main task is programming, and so, putting users' hints (that is, tagging) for weaving modularized documents is the additional issue. At the same time, however, documentation will help developers to understand the behavior of programs as long as they write documents on each programming modules. Therefore, our tool automatically weaves documents and does not require any additional tags. Instead, woven result is not so accurate. When software such as a library and framework is published, modularized documentation has to be precisely released for explaining how to use the APIs. Since wrong woven results are not acceptable at this time, developers have to put lots of tags to control weaving of documents correctly. After software is released, the part of implementations will be refactored again and again through several versions of it. In addition, some specifications might be also added, modified, or removed. Thus, tagging at this time should be less than that when released not so as to be affected by refactorings. The accuracy of woven result should be also relatively high. Although the tags are different from the one that are written when software is published, these two kinds of tags can be written for the same documentation at the same time. The important thing is that firstly written tags when software is released can be easily exchanged when needed. This is because it is difficult to predict which programming module might be refactored in advance. The only thing developers have to do is just replace the tags. They do not need to change modularized documentation themselves.

AspectScope

Aspect Oriented Programming enables to modularize concerns that crosscuts classes in Object Oriented Programming. The pointcut and advice mechanism that AOP languages such as AspectJ [48] provide allows developers to combine a module to a special module, called an aspect, without explicit method calls. This is useful to implement certain crosscutting concerns as a separate module. An aspect is implicitly invoked when a thread of control reaches some execution points in the other module. Those execution points are selected from the predefined set of points by the language.

However, this property of AOP makes it difficult for developers to understand the behavior of a module as long as they are looking at only the source code of that module. When one module is executed in an AOP language, other modules might be implicitly invoked from that module. The behavior might be changed by the deployment of other modules (*i.e.* aspects). Therefore, AOP languages require a whole-program analysis for understanding a program. Although module interfaces in AOP are hard to see, it should be possible to improve the visualization by a programming tool so that developers can more easily see the module interfaces under the current deployment of aspects. This would hopefully give better impression of AOP to the developers, who want to reason about their programs at a module level.

Therefore, we provide AspectScope to improve the understanding of programs developed in AOP. This is the viewer plugin working in Eclipse IDE. In its viewer, members such as methods and fields are shown which members are extended by aspects in its outline viewer. To show the concrete behaviors of these members, AspectScope also provides document viewer. This document viewer weaves documents of a method and advices that will be woven when the method is invoked. The only thing developers have to do is to write each document on methods and advices. The document viewer also analyzes the pointcut definition of the advice that extends the behavior of a method, and generates automatically the translated description on its document viewer. Through this description about the pointcut, developers can understand when the advices will be invoked. Note that the developers who uses our document viewer do not need to understand aspect definitions because the document viewer hides the constructs of aspects such as advices and pointcuts, which might be difficult to learn for OO programmers. The developers that can define aspects should write documentation of these advices and pointcuts.

CommentWeaver

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. 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, we provide 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. CommentWeaver provides lots of tags to control the weaving of modularized documents precisely. The woven results can be viewed as the API documentation in HTML files. Developers have to write these tags that specify concrete programming structures.

Universal AOP

Once software is released, it is ideal that rewriting documentation against refactorings is not quite often. Although CommentWeaver provides tagging system that specifies concrete programming structures such as names of super class and caller methods to other methods, this syntactic-based weaving of documents needs rewriting of documentation against refactorings.

To address this problem, we provide a documentation tool named *Universal AOP*, which enables the weaving of documents by using the grammar and semantics of the natural language. As CommentWeaver does, Universal AOP also provides the pointcut to weave modularized documents. However, the pointcuts do not specify the concrete programming structures but the natural languages in documentation. The pointcut of Universal AOP is defined with respect to the grammatical subject, verb, and object of the documentation sentences. Thus, Universal AOP enables semantic-based weaving of documents because this pointcut selects the semantics of programming specification. As another benefit of semantic-based weaving comes from that it is free from concrete programming structure. This means that developers can construct a view for modular understanding of programs through documentation whatever languages they use. The only thing developers have to do is just to write documentation on each programming modules. One of the reason why Universal AOP is free from concrete language constructs is that OOP has been widely used in industry and there is lots of hoard of using OOP languages even though new programming paradigms such as AOP have been developed in academia. In addition, the transition of major programming paradigm in industry will be very slow especially in the industrial environment that developers are familiar with OOP. Therefore, we provide a new paradigm for programming understanding by not introducing new programming constructs, that is, documentation.

Let us consider a software company wishing to benefit from a new programming paradigm that requires the use of new language extensions. This new paradigm will require new concepts and language constructs to be introduced into the company's business processes and code base. This can pose both a significant risk and cost to the company due to the associated learning costs but also due to the risk of failure and products not being delivered to specification nor to schedule. Furthermore, legacy software assets of the company may have to be re-implemented using the new approach to remain compatible. As a consequence of this, organisations are usually very slow and unwilling to adopt new programming practices. However, what if the new programming paradigm can be adopted without these risks? What if the programming paradigm does not require new language constructs to be introduced? What if the programming paradigm can be applied without change to legacy code? We propose that the modular views offered by these programming paradigms should be decoupled from the underlying implementation so that the benefits of these advancements can be quickly and safely applied. In order to demonstrate the advantages of our approach, we take Aspect-Oriented Programming (AOP) as a case-study. AOP is ideally placed to demonstrate our approach as it is a development technique that is currently relevant in terms of its position on its "hype-curve"; industrial organisations are interested in adopting it but are faced with the problems described above.

Aspect-Oriented languages are specifically designed to modularise scattered and tangled crosscutting concerns. A number of existing languages (most notably Java and C) have been extended with dedicated programming constructs to achieve this. These extensions allow crosscutting concerns to be represented within aspects and allow their composition via pointcut specifications. To take advantage of these concepts, developer's have to *learn a new (or extended) programming language.*

1.3 Position of this thesis

The position of this thesis is that modularization of documentation at the design and implementation phase has been considered because documentation is often crosscutting along programming modules. The important thing is that documentation cannot be necessarily written modularly even if source codes are well modularized. Since this notion has not been much discussed ever, we considered how to write documents that are suited with the modularity given by programming languages. In history, programming languages have been developed for better modularity such as the procedural programming, the object orientation, and recently the aspect orientation. On the other hand, the way of describing documents has not been changed, especially at the design and implementation phase. In the other phases prior to the design and implementation phase, there are several researches about modularly describing documents such as ARCADE [76] and RDL [24]. They enable to construct modular documents for the requirement engineering. However, in requirement engineering, there is no consideration about the relation between programming languages and documentation because program has not been implemented at this phase.

Another position of this thesis is that we provide the documentation tool not only for documentation but also for programming. The main purpose of modularizing implementations is to put source codes in order. As a result, the modularized implementation brings in the maintainability and reusability. We consider that this purpose need not be necessarily accomplished by programming languages. Instead, documentation can undertake this role as long as the classification of source codes can be perceived as *concerns.* Programming languages provide an explanation of a method semantics through its syntactic constructs. On the other hand, documentation also explains the specification of the method. Thus, each of them potentially has the same ability to make developers perceive concerns. To delegate the part of the role to represent concerns, we developed a documentation tool that is available on top of OOP. Our tool can be substitution of any AOP languages: documents explain crosscutting concerns that cannot be separated in OOP. The underlying reason why AOP languages have to be replaced with our tool is that AOP languages have not been widely used in industry ever since AOP was first developed more than ten years ago. The difficulty to learn the paradigm of AOP and its languages may be one of the reasons. However, the notion that AOP brought about is still very significant. Separating crosscutting concerns into aspects increases the modularity. Thus, we consider how the same effects of AOP become available without introducing new paradigms and language constructs. Documentation can provide modular views of crosscutting concerns, and in addition, writing/reading documentation does not require any additional knowledge.

1.4 Structure of this thesis

The next chapter declares the problems that we are tacking on. Then, the later three chapters propose each tools for modularizing documentation at each software development phase. The summary of this thesis is as follows.

Chapter 2: Documentation of Modularization

This chapter presents that there are several existing researches but they are only partly tackling on our aims: modularization of documentation. Then, the properties that should be achieved for our aims are considered, and our ideal solution is also presented here.

Chapter 3: AspectScope

To support the understanding of the whole program behaviors that are written in Aspect-Oriented Programming, we propose a viewer tool that are useful at the implementation phase. It provides documentation generated from classes and aspects that affect these classes. We implemented a tool as an Eclipse plugin for AspectJ.

Chapter 4: CommentWeaver

To generate precise API documentation when software is released, we propose a documentation tool, which allows developers to write documentation as a modular fashion. It provides several tags to compose modularized documentation. The woven result can be viewed in the HTML format.

Chapter 5: Universal AOP

To make modularized documentation robust through refactorings of several versions of software, we propose another documentation tool that are less fragile against refactorings. It provides semantic-based composition rule into composing each documentation, and it is free from concrete programming structures.

Chapter 2

Modularization of Documentation

Through software development, writing documents is one of the significant tasks although writing source codes is the main pillar to construct software. At the requirement and specification phase, documentation is crucial to define how the software works. On the other hand, at the implementation and design phase, documentation is not mandatory. However, documentation helps readers' understanding of the software. For example, developers themselves may need documentation that had been written by themselves to understand the implementations. 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 [15].

Even though writing documents is a burdensome task, a technique for documentation has not been ever considered much about. However, the quantity of describing documents is unexpectedly large, and the quality of documentation is also required. For example, the size of the Javassist library (version 3.6) is 53,477 lines of code (LOC) and 9,512 of 53,477 lines are doc comments for Javadoc (18 %). In case of Java 6, the code size is 2,038,855 LOC and 575,887 of them are doc comments (28 %). As another

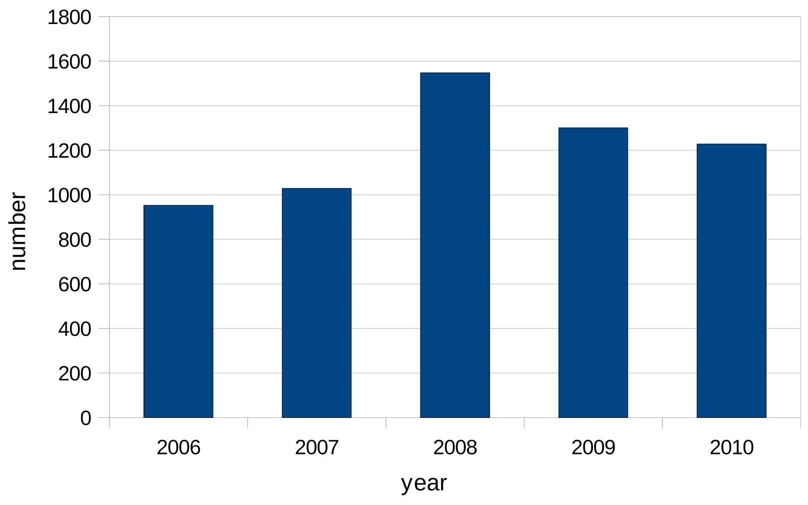


Figure 2.1. The number of bug reports about documentation in Eclipse

example, misunderstandable documentation led a trouble. One of bug reports to Javassist required to explain detailed behavior of an API method [4], because insufficient description prevented user programmers to run his software precisely.

In addition, we conducted a survey on the number of the document-related bug reports found in the bug report site of Eclipse [3]. All the related reports such as Help documents and Wiki documents were counted. As shown in Figure 2.1, in most years, more than 1,000 bug reports about documents were posted. The average number of these reports is 1,212 per year, roughly 3 % in all bug reports. This is an astonishing result to consider that Eclipse is one of the widely-used software that have been maintained after published as an open source software in 2004.

We also investigated the details of 500 bug reports that are related with doc comments. As shown in Figure 2.2, bug reports were categorized by four kinds. The major report was about insufficient documents. For example, a bug report mentions to developers that they should add documentation because users don't seem to understand how to use the API correctly. The ratio of such kind of reports was 70.6%. The second most report was about incorrect documents. The reports that mentioned that documentation does not precisely explain the specification of methods were counted. The ratio of these reports was 18.7%. The next most report was about out-of-date documents, which were not updated with updates of specifications. Since

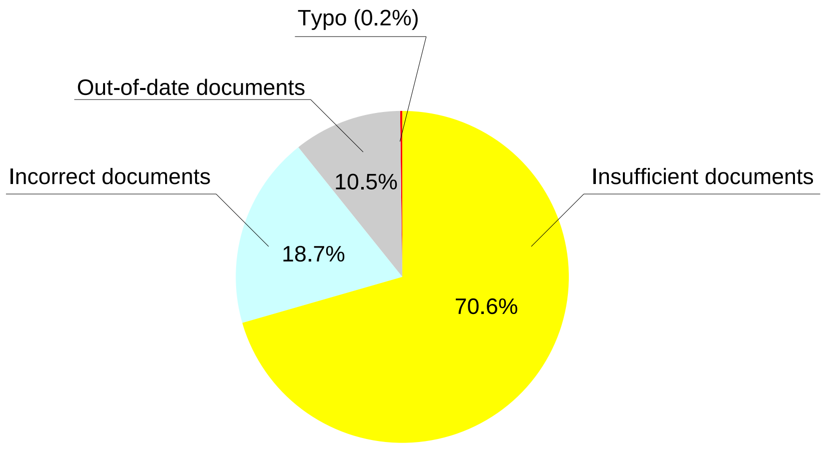


Figure 2.2. The number of bug reports about documentation in Eclipse

these documents have not explain the current behavior of a program, they should be revised. Although these kinds of reports have overlaps with the insufficient documents and incorrect documents, we counted the number separately. The ratio was 10.5%. The reports that mentioned typo was only 0.2%.

We consider that one of the reason for these bugs comes from the difficulties of maintenance of documentation as well as writing documentation. Documentation for OOP programs becomes more difficult because call hierarchies of methods tend to be deeper, and , all these methods basically need documentation. Note that, since OOP provides a mechanism of encapsulating a function, the encapsulation *hides* the details of that function. Thus, an explanation that tells what the encapsulation does is needed such as a long method name and documentation. What is worse, the documentation of the same call hierarchy have a lot of overlaps in their descriptions, which seem to be too complicated to manage them for a change.

Programming paradigms have been invented for higher modularity to increase maintainability and reusability. Also, lots of programming languages for each paradigm have been considered and some of them still have been widely used in academia and industry. On the other hand, the way of writing documentation have not been so much changed. In the implementation and design phase, since documentation have to be written along with source codes that are well modularized, more efficient way of writing documentation should be necessary.

2.1 An ideal description for documentation

Then, what is the ideal description to write documentation? We consider that one of the best solutions is to modularize documentation. Here, modularizing documentation means writing a description on its programming module just once. When a programming module is modified, it will become easy to modify its description along with its programming modification. The second factor for the ideal description is that the natural language is available in documentation. This is because we consider that learning brandnew way of writing must be another burden to construct software. The two main benefits delivered by modularity are improved software reuse and maintainability. Reuse is achieved due to relative physical independence between modules, which can be subsequently used without change in different systems. Maintainability is achieved due to localisation of code, which can be viewed in one place, and changed locally, thus reducing the scope of the change. While the reuse property requires physical mobility of the code, maintainability can often be satisfied via on-demand localisation of the relevant code, irrespective of its physical location.

In lifetime of any system, the main development tends to be a one-off event, whereby the initial system is delivered. For this initial system implementation the existence of a suitable programming language and its correct choice is of paramount importance. However, in accordance with many studies [16] more than 80% of costs within the system life time will likely be committed to its maintenance. Once delivered, the physical implementation of the softer system cannot be chugged, but its maintainability can still be facilitated. We advocate that this facilitation about writing documentation will be achieved through modularized one. However, this natural way of writing cannot be permitted on the current documentation tools and systems.

If documentation cannot be modularized along with one concern, the descriptions of the documentation might contain several descriptions for several concerns. This situation can be said as the documentation is tangling. For example, in Figure 2.1, the text starting with “Once this method is called” is written in documentation of three methods writeFile(), writeFile(String), and toBytecode. Since the basic behavior of two writeFiles is to convert a class definition into a file, the documentation of writeFile is tangling because of the text starting with “Once this method is called”. Note that the actual implementation for this description is realized in the toBytecode method. Therefore, the text starting with “Once this method is called” should be only written in the documentation of toBytecode. Also, a description might

```
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.
  *           :
  */
 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.
  *          :
  */
 public void toBytecode(DataOutputStream out) throws .. {
     throw new CannotCompileException("not a class");
}}
```

Figure 2.3. Scattering and tangling documentation

be written in several documentation. One of the example is the description about the toBytecode method. The text starting with "Once this method is called" is scattering among two writeFile methods.

Another factor for the ideal documentation writing is that the natural languages are available to write documentation.

2.2 Existing tools for modular documentation at the implementation phase

Unfortunately, existing tools are not sufficient for our purpose; to write documentation as a modular fashion along with the modularization techniques for programming languages, and the description is realized with the natural language.

Javadoc

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 [15]. In Java, the Javadoc tool [67] helps to write API documentation. Javadoc tool processes documentation that are written in Java source codes into HTML files. It enables writing API documentation as comments directly embedded in program source files as shown in Figure 2.1. 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. However Javadoc does not enable developers write doc comments modularly. Javadoc tool provides its special tags to make specific statements bold in the generated HTML file. For example, the @param tag takes two arguments; one is a parameter name and another is its description as shown below.

```
@param fileName   A file name that are passed through ...
```

The @param tag is processed through Javadoc tool to have emphasized note in a html file. Javdoc tool also provides the @see tag that takes an argument to append a hyper link to the argument in the generated html files. Suppose that the @see tag is used in the doc comment of the writeFile method as follows:

```
@see CtClass#toBytecode(DataOutputStream)
```

In the generated html file for the CtClass class, the API documentation of the writeFile includes the hyperlink to the API documentation of the toBytecode method. The @see tag is useful to make user programmers notice some relationship between two methods. Although to link two methods by hyperlinks might be sufficient in some cases, this way of representation will be problematic when user programmers want to know which statements in a linked method are most relevant to a method. Note that what @see does is just putting hyper links between two methods and API documentation of these two methods might have lots of unrelated documentation about these two methods. Therefore, developers have to read and find which statements are important to understand the relation of two methods by themselves. This is also a burdensome task.

Then, some developers may think that aspect-oriented programming (AOP) [48, 62] makes programming modularity better and so using AOP allows developers write modular documentation as well as modular programming. AOP enables to separate crosscutting concerns that cannot be modularized by OOP techniques. In addition, a documentation tool for AspectJ, which is one of the most famous AOP languages, is available as the name of Ajdoc tool [71],which is another Javadoc-like tool for AspectJ. It enables us to attach a doc comment to an advice body in an aspect. However, Ajdoc is still problematic. Suppose to reimplement the program shown in Figure 2.1 by using AspectJ. In Figure 2.2, an aspect named BytecodeDescribing is defined and the method body of the toBytecode method is moved into an after advice of it. In the pointcut of this after advice, execution of writeFile(String) is specified so that the advice will be invoked implicitly when writeFile(String) is invoked. Note that the whole behavior of this program does not change at all. In the doc comment of the writeFile(String) method, the text starting with "Once this method is called" is removed. Thus, now the doc comment of the writeFile(String) method is only described just about this method and tangling problem is addressed. However, there are a few problems remained. Firstly, generated API documentation through Ajdoc tool does not provide dedicated descriptions to user programmers. Ajdoc tool generates the API documentation of aspects and the pages between the affected classes by aspects and these aspects are connected by using hyper links. Therefore, user programmers step into a link to read the aspect behavior that will affect the class behavior. However, there is no guarantee that aspects should be always defined as public; some aspects should be keep private. For example, if a caching aspect that keeps class definitions for frequent access of them is defined, this aspect should not be declared as public because user programmers do not need to know that this program is used a caching system. Currently, Ajdoc tool cannot handle this problems. The second problem is that the influence of an aspect to API documentation may be spreaded though the call-graphs of the method that is selected by a pointcut. On the other hand, Ajdoc tool cannot manage the propagation of the influence of an aspect through caller-callee relations. In the example of Figure 2.2, the doc comment of writeFile() still contains the text starting with "Once this method is called" because it invokes the writeFile(String) method that will be affected by the advice in the BytecodeDescribing aspect. Ideally, this text should be also removed from the doc comment of the writeFile() method.

There is still one big problem; is it really the best solution to implement always by AOP? Proliferation of AOP for the purpose of modularizing documentation must make programmers confused. While AOP languages such as AspectJ are useful to separate crosscutting concerns, the concerns that are once separated from a viewpoint are difficult to be reimplemented from the other viewpoints. Note that lots of patterns for separating concerns can be possible in one impementation. Thus, sometimes AOP languages will make programs difficult to understand, and documentation as well. Therefore, there is a certain number of programs that do not need to use AOP languages. For example, although an aspect is introduced in Figure 2.2, another implementation in Figure 2.1 is already well modularized and AOP-version does not seem to be necessary in this case. OOP techniques for encapsulating programs such as inheritances and methods may work well in this example.

Literate programming

Although literate programming is not a documentation tool that promotes modularization of documentation, it is the important programming style when we consider the relation about programming and documentation. WEB [53], CWEB [55], and FWEB [11] are languages based on the concept of *literate programming* [54], 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

```
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.
  *           :
  */
 public void writeFile(String directoryName) throws .. {
     DataOutputStream out = ...;
          :
 }}

public aspect BytecodeDescribing {
 /**
  * Converts this class to a class file. Once this method is
  * called, further modifications are not possible any more.
  *          :
  */
    after() : execution(void CtClass.writeFile(String)) {
             :
    }
}}
```

Figure 2.4. Revised documentation in AspectJ

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.

The important thing is that literate programming not only promotes to putting documentation into source codes but also values to write documentation as well as to write programs. Note that literate programming is not a technique to modularize documentation, but its documentation-driven style of programming may lead to the good style for writing well modularized documentation. Documentation in literate programming contains concrete fragments of the Pascal code that is divided into several procedures. Figure 2.2 is an example of literate programming. In this program, documentation is mainly described and program named sample is divided into small pieces of code. Since the description about loop procedure is mentioned in the next paragraph, the procedure itself is also implemented in this paragraph. In program sample in paragraph 1, the reference to the loop procedure is in the third line. Another procedure named Printing is referenced in the fourth line and instead the detailed description is written in the third paragraph and concrete program fragment for printing is defined there. When developers want to create only descriptions of paragraph 1,2, and 3, they can use the WEAVE processes to generate $\TeX$ sources of them. On the other hand, when developers want to get only Pascal source codes, the TANGLE processes to compose separated pieces of codes such as loop procedure and printing procedure.

The problem of literate programming is that implementing by using OOP is an open issue. Since documentation is mainly described and program are divided into small pieces of codes along each description, and they are putting into each paragraph, this style of documentation seems be difficult to be introduced in OOP notion. For example, a class structure should be needed as follows.

```
TSample = class(TObject);
Public
  M: Integer;
  N: Integer;
      :
```

Thus, The challenging thing is whether documentation will be enough to represent objects or not. Currently, no research has been proposed for this purpose.

```
1. Introduction. This is a simple example to explain how to write
 literate programming. Here is an outline of a sample code:

  program sample;
    begin
       <Loop 2>
       <Printing 3>
    end

2. Loop procedure does something needed in a loop.
  <Loop 2> ≡
  i ← 0;
  while i < 100  do
    begin
       {do something}
    end;

3. Printing procedure just prints out some string characters.
  <Printing 3> ≡
    print('Loop is finished');
```

Figure 2.5. Literate programming

2.3 Other documentation-related systems

There are several tools and languages that are related to modularization of documentation. In addition, even though documentation cannot be necessarily modularized, there are alternative techniques that contribute to understand OOP programs as well as documentation does.

RDL

Requirements description language (RDL) is proposed for modularizing requirements and their concerns in the requirement phase [24]. RDL has a mechanism to select and compose documents by using semantic-based pointcuts. A semantic-based pointcut selects verbs and nouns in the natural languages. Therefore, unlike other language constructs such as ARCADE [76], the semantic-based pointcut is less fragile than other requirement languages [23]. Figure 2.3 shows an example of describing a requirement in ARCADE. The aspect tag modularizes the display updating concern from the base requirements that are described from the Viewpoint tag. The Composition tag defines the rule of composition of these two requirements. Note that the

```
<Viewpoint name="User">
   <Requirement id="1">The user draws lines, triangle, and
                       rectangles on the figure editor.
   </Requirement>
   <Requirement id="2">The user can move figures on the
                       figure editor.
   </Requirement>
        :
</Viewpoint>

<aspect name="ObserverAspect">
   <Requirement id="1">The system updates the display of
                       the figure editor.
            :
   </Requirement>
</aspect>

<Composition>
  <Requirement aspect="ObserverAspect" id="all">
    <Constraint action="provide" operator="for">
       <Requirement viewpoint="User" id="1"/>
    </Constraint>
  </Requirement>
    :
</Composition>
```

Figure 2.6. Composition with ARCADE

composition is defined by using syntactic constructs such as the id number and the viewpoint name. On the other hand, Figure 2.3 is an example of writing a composition definition in RDL. The important construct for the semantic-based pointcut is the Base element. In this element, the relationship is specified as *write* and the object as *file* to capture statements that have the verb *write* and the noun *file*. For example, text starting with "Writes a class file" will be captured in this pointcut. In addition to the relationship and object, the subject is also available to specify a subject in sentences. On the other hand, other languages such as ARCADE specify concrete tags and attributes that are appended to each requirements and their concerns, and these syntactic based pointcuts may be more fragile when a requirement is changed through several versions of software.

RDL also allows to use synonyms of verbs to capture correctly semantic of sentences. Moreover, when developers want to capture broader meaning of a verb, they can use verb categories that means kind of verb meanings.

```
<Composition name="sample">
  <Constraint operator="enable">... </Constraint>
  <Base operator="metBy"> relationship="write" and object="file"</Base>
  <Outcome operator="fulfilled"/>
</Composition>
```

Figure 2.7. Example of RDL descriptions

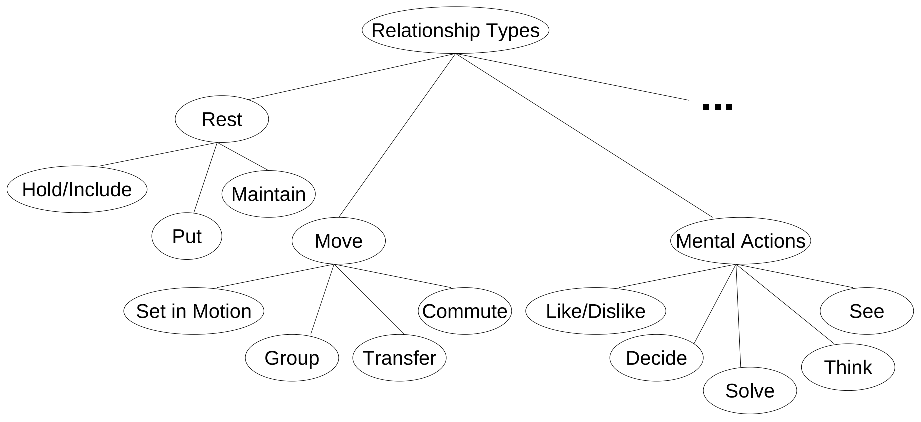


Figure 2.8. Verb categories in RDL

For example, the verb run and crawl will be categorized as one group of the same meaning of verbs (Figure 2.8).

However, there are several differences of how to write documentation between at the implementation phase and requirement phase. One of such differences is that there are many similar documentation at the implementation phase. Especially, libraries provide lots of similar APIs to increase the usability of user programmers as shown in Figure 2.1. Documentation at the implementation phase also has concrete name references. For example, the @see tag is one of the Javadoc tags as mentioned before and it is often used to refer to a concrete method name to put a hyper link in the API documentation. RDL cannot be applied to such demands happened at the implementation phase.

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 [49]. 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 or Universal AOP is requisite. The terminologies for categorizing classes are introduced as follows:

- Specified definition
  Classes that are published as API of the library
- Implementation-specific definition
  Classes that construct the library but do not appear as API
- Portable definition
  Classes that are defined by user programmers of the library, and that only depend on the specified definitions.
- System-defined definition
  Classes that consist of either specified or implementation-specific definitions
- User-defined definition
  Classes that are defined by user programmers of the library

There are mainly four concrete proposals to implement good frameworks. Firstly, implementation-specific class should be allowed to be defined between specified classes. For example, suppose that two specified classes are defined: the Shape class is a super class, and one of its subclasses is the Rectangle class. Even if the Rectangle-base class is defined as the subclass of the Shape class and the super class of the Rectangle class, this design should be allowed to user programmers. Secondly, framework user should be able to promote methods that were once defined in a specified class into an interposed super class. Note that this promotion of methods does not reflect directly to user programs. For example, if framework developers want to promote a method that is defined in the Rectangle class, the mehtod can be moved into the Rectangle-base class. Next, There should be no portable classes that inherit instance fields of specified super classes. If encapsulated is well conducted, that is, instance fields cannot be accessed only though methods, framework developers do not need to pay attention to this rule. Finally, user programs should not redefine any specified classes and methods. This rule is natural because frameworks each have their constraints how to use thier frameworks, and user programers should not break these constraints.

JML

The Java Modeling Language (JML) [57, 58] is a language for the design by contract. Although this is not a documentation tool that processes the API documentation, JML commonly uses the comment-style description in source codes. On the other hand, Eiffel [63, 64] and D language [19] have a mechanism to manipulate the design by contract as the part of the languages.

As an example of JML description, following is a contract of the Person class that defines the name field to represent its person's name and the score field to represent his/her score of a class. Since the name should not be null, JML provides non_null to satisfy the contract. One of the methods defined in the Person class is the addScore method, which adds the number of argument to the score. As shown in the comment of the addScore method, this method ensures that the argument should be more than zero.

```
public abstract class Student {
  private /*@ spec_public non_null @*/
    String name;
  private /*@ spec_public @*/
    int score;

      :

  /*@ also
    @ requires score >= 0;
    @*/
  public void addScore(int score);

  /*@ also
    @ requires n != null && !n.equals("");
    @ ensures n.equals(name)
    @ && score == 0; @*/
  public Person(String n);
}
```

As an extension of JML, Clifton *et.al.* proposed aspect-oriented version of JML [28, 29]. The most distinctive feature of their system is to manipulate two types of aspects: *assistants* and *spectators*. The assistant means an aspect that changes the semantics of the base classes, while the spectator means an aspect that merely observes class behaviors. The role of spectators resembles pure aspects [85]. For example, a non-functional aspect such as a logging aspect is a spectator that does not change the semantics of any other classes.

As an example of assistants, the Contract aspect below can be said as an assistant. The before advice can change the specification by throwing the IllegalArgumentException when the argument of the setter methods in the Point class does not fulfill the given conditions. When developers do not want to change the semantics of a class, they can declare that the class must not be influenced by an aspect by using the accept clause, which takes the name of an aspect in its argument. For example, Point class accepts the Contract assistant by declaring *accept Contract* in the Point class.

```
aspect Contract {
   before(int x):
        call(void Point.setX(int)) && args(x) {
      if (x < 0 || 100 < x)
        throw new IllegalArgumentException();
   }

   before(int y) :
       call(void Point.setY(int)) && args(y) {
      if (y < 0 || 50 < y)
        throw new IllegalArgumentException();
   }
}
```

To reduce the burden of writing accept clauses in the base classes, accept maps can be used. An accept map allows developers to write specifications of acceptance in one place. This module can resolve not only tangling codes between class codes and accept clauses, but also scattering of accept clauses. An example aspect map is as follows.

```
package figures;
Point {
   accept Contract;
}
Line {
   accept Contract;
}
* {
   accept DisplayUpdating;
}
```

The Line pattern in this example says that Line class accepts Contract assistant. The next module starting with wildcard * indicates that every classes accept the DisplayUpdating assistant. Therefore, the Line class accepts both the Contract and the DisplayUpdating assistants.

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 [65, 59], the typestate checking [32, 14], and the FUSION analysis [45]. The main difference from typestates is that FUSION can abstract relationships that framework users can manipulate. The FUSION analysis is useful for specifying framework constraints such as the semantics constraints between multiple objects.

FUSION is constructed under careful observations in frameworks. The constraints in the framework involve multiple classes and objects. This is because, in OOP, a method usually invokes several other methods, which are defined in different classes. Secondly, these constraints in frameworks are often extrinsic. In their paper, the ASP.NET ListControl class is used as an example. Framework users can define subclasses of the ListControl class to extend the framework. The ListItem class is one of the framework classes, and it has a constraint about the usage of its object. The problem is that the user defined subclasses have to check the constraint of the ListItem class. However, since these subclasses are defined by framework users, the ListItem should not be responsible for enforcing the constraint. In this paper, the difficulties of documentation are also written in case of describing such framework constraints. In my opinion, these constraints should be described in the super class, because when framework users define a subclass to extend the framework, they may read the documentation of the super class to know how they can extend the base behavior and what kinds of constraints exists. Finally, the constraints in the framework have semantic properties. This means that framework users have to pay attention to the specification of the objects such as ordering of invocation of the objects.

To abstract relations between objects, framework developers can append annotations to each method definitions. For example, when a developer need to put a relation between the set object and the element object, they can define the relation as follows:

```
@set({set, item}, ADD)
```

On the other hand, if they want to remove this relation, they can define as follows:

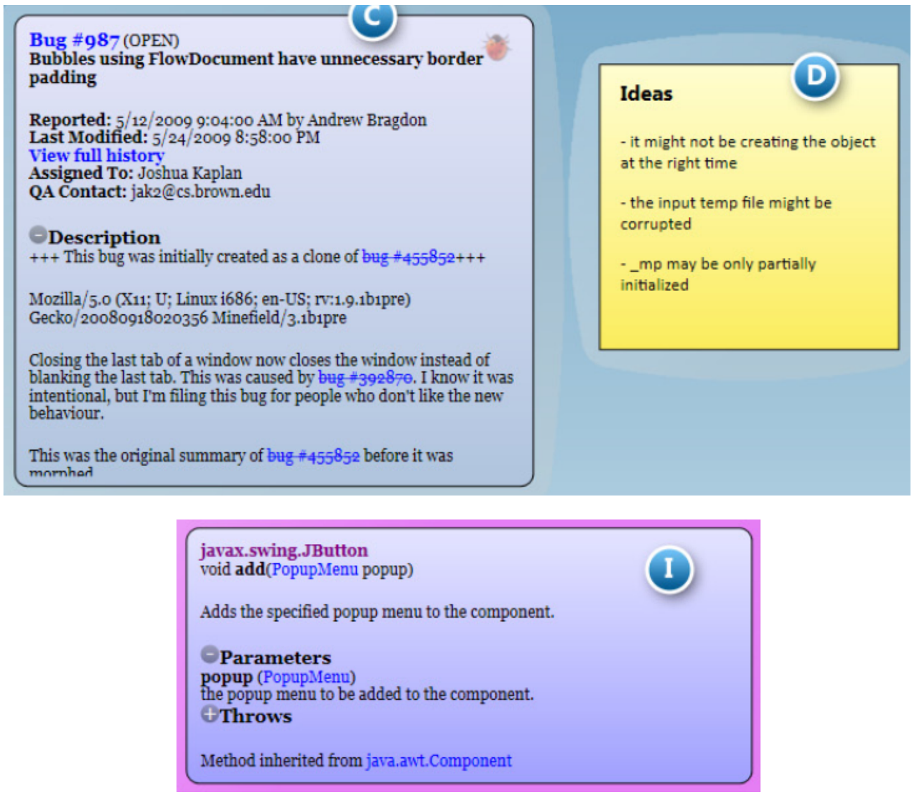
```
@set({set, item}, REMOVE)
```

In the argument of the set relation, the wildcard * is available. The target and the result are also available in the argument of relations. When an annotation that uses the target in the argument is defined on a method, the target represents an object that invokes the method. In case that result is used, it represents the return value of the invocation of the method by the object.

CodeBubbles

Code Bubbles [18] is the new IDE, which are developed based on the ideas that large scale programs does not fit into display in nature, viewing overlapped editors needs manual interaction by users. Editing codes in Eclipse IDE often needs large window display to show all the related codes in practice. If there is no large window display, developers have to duplicate several editors, and this will result in thier time consuming development. On the other hand, Code Bubbles will decrease the time developers have to spend for browsing source codes, debugging codes, and reading documentation. To help these users' activities, Code Bubbles provide flexible editable fragments on its IDE. Debugging is partly available as well as editing source codes. The notion of the file does not exist in Code Bubbles IDE. Thus, developers use only bubbles metaphor by implementing codes. There have been several editors that are based on non-file style programming. The most famous editor is the one for Squeak [44], which does not show file structure to users unlike Eclipse IDE. There are lots of editors that share similar notion of non-file style programming [77, 46, 83].

In addition, this IDE enables to build heterogeneous working sets from many kinds of resources: source codes, Javadoc, and other notes (Figure 2.9). Therefore, Javadoc comments are shown when required on its IDE, and developers can understand the specification of methods before looking into their implementations. Bug reports and notes can be append as bubbles on the IDE. However, crosscutting concerns of Javadoc comments cannot be resolved on this IDE, although Code Bubbles seem to make it easier to write documentation than the typical file style editor such as Eclipse editor does. Currently, Code Bubbles only enable language level support for AOP, and so, there is no IDE support to separate crosscutting concerns in methods as well as crosscutting concerns in documentation. However, modularization of documentation can be still introduced in Code Bubbles: a Javadoc bubble



Bug #987 (OPEN)
Bubbles using FlowDocument have unnecessary border padding

Reported: 5/12/2009 9:04:00 AM by Andrew Bragdon
Last Modified: 5/24/2009 8:58:00 PM
View full history
Assigned To: Joshua Kaplan
QA Contact: jak2@cs.brown.edu

Description
+++ This bug was initially created as a clone of ~~bug #455852~~+++

Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1b1pre) Gecko/20080918020356 Minefield/3.1b1pre

Closing the last tab of a window now closes the window instead of blanking the last tab. This was caused by ~~bug #392870~~. I know it was intentional, but I'm filing this bug for people who don't like the new behaviour.

This was the original summary of ~~bug #455852~~ before it was morphed

Ideas

- it might not be creating the object at the right time

- the input temp file might be corrupted

- _mp may be only partially initialized

javax.swing.JButton
void **add**(PopupMenu popup)

Adds the specified popup menu to the component.

Parameters
popup (PopupMenu)
the popup menu to be added to the component.
Throws

Method inherited from java.awt.Component

Figure 2.9. Bubbles related with documentation in Code Bubbles

of an aspect will be merged into the other Javadoc bubbles of the target methods, or our tagging system can be append into methods in bubbles to modularize documentation. In Code Bubbles, editing the methods in a call-hierarchy will be easy by showing all methods as bubbles that show these methods have caller-callee relations. Thus, editing documentation in one call chain will be easy. This might be efficient to reduce bugs of insufficient documentation shown in Figure 2.2, because one of the main reason of this bug may come from the complexities of method hierarchies, and Code Bubbles will ease this problem.

2.4 Summary

In this chapter, we discussed the requirements for modularizing documents. Ideally, documents should be separately written with its programming mod-

	Modularization of documents	Written as comments
Javadoc		Yes
Literate programming	Yes/No	

Table 2.1. Comparison of existing tools

ule as comments n source codes. However, as shown in Table 2.1, neither of them fulfill our requirements. Documentation tools such as Javadoc do not provide a solution to write documents in modular fashion although documents can be written in source codes as comments. On the other hand, literate programming can provide a way of modular description of documents. However, it does not allow to write documents as comments. This hinders applicability into other programming paradigms such as OOP.

Chapter 3

AspectScope

The pointcut and advice mechanism of Aspect-Oriented Programming (AOP) languages such as AspectJ [48] allows developers to combine a module to a special module, called an aspect, without explicit method calls. This is useful to implement certain crosscutting concerns as a separate module. An aspect is implicitly invoked when a thread of control reaches some execution points in the other module. Those execution points are selected from the predefined set of points by the language.

However, this property of AOP makes it difficult for developers to understand the behavior of a module as long as they are looking at only the source code of that module. When one module is executed in an AOP language, other modules might be implicitly invoked from that module. The behavior might be changed by the deployment of other modules (*i.e.* aspects). Therefore, AOP languages require a whole-program analysis for understanding a program.

To address this problem, several programming tools for AOP have been developed. One of the most popular tools is AJDT, AspectJ Development Tools of Eclipse IDE [1]. It automatically performs a whole-program analysis and visualizes the crosscutting structures in the program according to the result of the analysis. The developers do not have to manually perform a whole-program analysis any more. However, AJDT does not seem to satisfy developers. Their claim is that they want to see *static* module interfaces for understanding their programs. Here, the module interfaces include the specifications of the behavior of the modules. Although AJDT automatically performs a whole-program analysis while a developer is editing a program, the visualization by AJDT does not much help the developer see the module interfaces. It does nothing except simply showing the join points where modules are combined with aspects. Even worse, module interfaces in AOP languages are never static or stable. It changes according to the deployment of aspects. In this sense, the module interfaces in AOP are essentially different from traditional ones.

Although module interfaces in AOP are hard to see, it should be possible to improve the visualization by a programming tool so that developers can more easily see the module interfaces under the current deployment of aspects. This would hopefully give better impression of AOP to the developers, who want to reason about their programs at a module level.

This chapter presents *AspectScope*, which is our programming tool for AspectJ. We have developed it for realizing our idea above. Like AJDT, it automatically performs a whole-program analysis and visualizes the result. However, it shows how aspects affect module interfaces in the program. It interprets an aspect as an extension to other classes and it displays the extended module interfaces of the classes under the deployment of the aspects. It thereby helps developers understand crosscutting structures in the program.

3.1 Modular Reasoning

The standard AspectJ support of Eclipse IDE, named AJDT [1], visualizes a crosscutting structure in an AspectJ program. This helps developers to reason about the program with a modular fashion despite the obliviousness property of AspectJ [38]. However, the help by this visualization is still limited and thus developers sometime feel that AOP makes modular reasoning difficult.

To illustrate the limitation of AJDT, we below show a refactoring process of a figure editor [50] as an example scenario. A figure editor is a simple tool for editing drawings that are composed of points and lines. Since a display of the tool must always reflect the current states of such shapes, any method that is declared in Point or Line class must call the update method in the Display class whenever that method changes the states of shapes. The update method redraws a display so that the pictures of all the shapes such as points and lines on the display will be updated. Figure 3.1 shows the AspectJ program of this figure editor. The concern of updating a display is implemented in an aspect.

```
public interface Shape {
  void moveBy(int dx, int dy);
}

class Line implements Shape {
   private Point p1, p2;
   public void setP1(Point np1) {p1 = np1;}
   public void setP2(Point np2) {p2 = np2;}
   public Point getP1() {return p1;}
   public Point getP2() {return p2;}
   public void moveBy(int dx, int dy) {
       p1.x += dx; p1.y += dy;
       p2.x += dx; p2.y += dy;
   }
}
```

```
class Point implements Shape {
  public int x, y;   // intentionally
  public void setX(int nx) {x = nx;}
  public void setY(int ny) {y = ny;}
  public int getX() {return x;}
  public int getY() {return y;}
  public void moveBy(int dx, int dy) {
      x += dx; y += dy;
  }
}

aspect UpdateSignaling {
  pointcut change():
      call(void Point.setX(int))
      || call(void Point.setY(int))
      || call(void Shape+.moveBy(int,int));

  after() returning: change() {
      Display.update();
  }
}
```

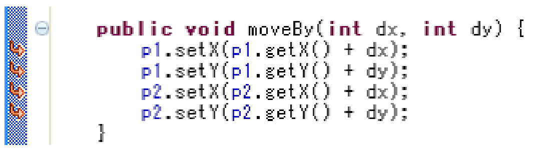
Figure 3.1. A figure editor implemented in AspectJ

We then perform simple refactoring. The x and y field in the Point class are intentionally public. Since this fact is obviously a weakness in information hiding, suppose that a developer changes these fields to being private. This change causes another change in the moveBy method in the Line class. The fields x and y on p1 and p2 are not accessible any more. The developer must change the moveBy method. Figure 3.2 shows the new revision of moveBy method.

Unfortunately, the refactoring has not finished yet. If the moveBy method is invoked, the developer will see that the display flickers. To understand this problem, the developer will have to investigate the whole program including aspects and find which join points are advised. Local investigation within the moveBy method or the Line class does not reveal the problem to the developer because of the obliviousness property of AspectJ. The lexical representation of the moveBy method does not contain any sign or symptom of being advised.

```
public void moveBy(int dx, int dy) {
    p1.setX(p1.getX() + dx);
    p1.setY(p1.getY() + dy);
    p2.setX(p2.getX() + dx);
    p2.setY(p2.getY() + dy);
}
```

Figure 3.2. The new revision of moveBy method



```
public void moveBy(int dx, int dy) {
    p1.setX(p1.getX() + dx);
    p1.setY(p1.getY() + dy);
    p2.setX(p2.getX() + dx);
    p2.setY(p2.getY() + dy);
}
```

Figure 3.3. AJDT indicates advised join points

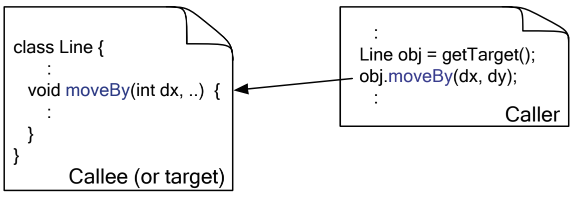


Figure 3.4. The caller side and the callee side

AJDT helps the investigation. It automatically performs a whole-program analysis and visualizes which join points are advised by an aspect. See four arrow icons at the left side of the source editor in Figure 3.3. The developer can notice that the four calls to setX and setY in the body of moveBy are advised by the UpdateSignaling aspect, which invokes the update method in the Display class to repaint the display and cause a flicker.

However, the help by AJDT is not sufficient. The developer, who saw Figure 3.3, would change the moveBy method so that only the first call to setX would be advised. Suppose that she changes the moveBy method to the following:

```
public void moveBy(int dx, int dy) {
  p1.setX(p1.getX() + dx);
  p1.incY(dy);
  p2.incX(dx); // incX() and incY() are not advised
  p2.incY(dy);
}
```

She also adds new methods incX and incY for incrementing the value of x or y. We assume that calls to these new methods are not advised. Unfortunately, this change does not stop a flicker. Although now only one join point (*i.e.*

a call to setX) in the body of the moveBy is advised, a call to the moveBy method itself is also advised. Therefore, each call to the moveBy method causes two successive invocations of the update method and they cause a flicker.

The problem here is that the developer cannot notice that a call to moveBy is also advised as long as she is looking at the source code of the moveBy method. AJDT does not tell her the fact unless she opens a client class of Line and then looks at a *caller-side* method of moveBy (Figure 3.4). If a pointcut is call, AJDT puts an arrow icon only at a method-call expression that calls the advised method. It does not show any indications at the *callee-side*. Note that a call pointcut selects join points at which method-calls are executed in a client class, while an execution pointcut selects join points at which method bodies are executed. Thus, to reach a right solution, the developer must manually perform a whole-program analysis to a certain degree and understand the crosscutting structure in the program. Then she must edit the aspect program so that the update method will be invoked only once for each top-level change of the state of the shape. The revised UpdateSignaling aspect is the following:

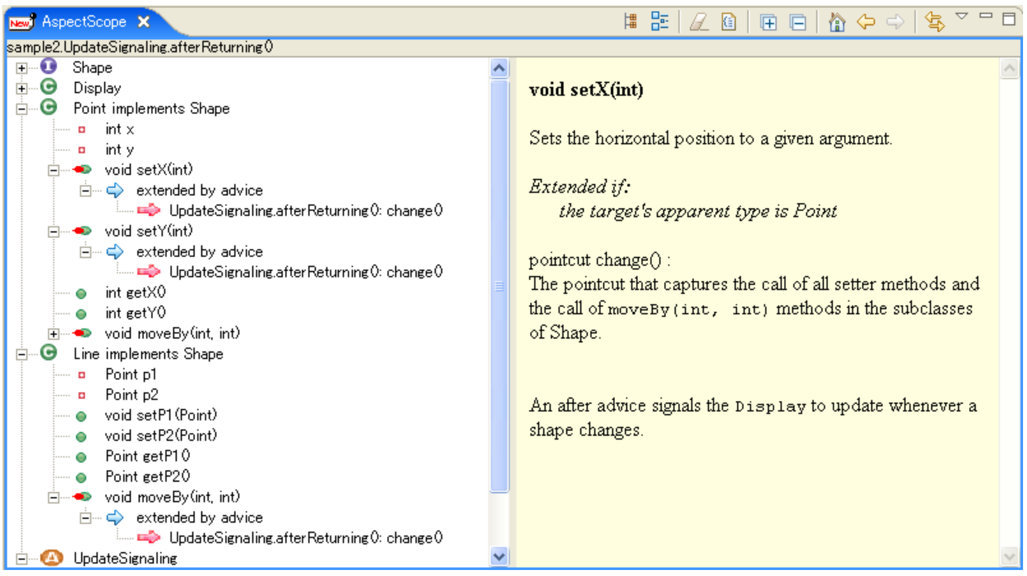
```
after() returning:  change() && !cflowbelow(change()) {
  Display.update();
}
```

Now the update method is invoked only when either setX, setY, or moveBy is called as a top-level call. Since a cflowbelow pointcut selects join points below the control flow of the specified join points, update is invoked only once for each call to the moveBy method. The developer does not have to add incX or incY to the Point class.

3.2 AspectScope

Although AJDT visualizes crosscutting structures in a program, it only indicates where a crosscutting structure joins other structures, that is, it only indicates join points in the source code. As we have seen in the previous section, this visualization is not sufficient to help developers understand crosscutting structures in their programs.

For better help, we have developed another programming tool for AspectJ. It is an Eclipse plugin named *AspectScope*. This tool visualizes cross-



```
AspectScope
sample2.UpdateSignaling.afterReturning()
⊞ Shape
⊞ Display
⊟ Point implements Shape
    int x
    int y
  ⊟ void setX(int)
    ⊟ extended by advice
        UpdateSignaling.afterReturning(): change()
  ⊟ void setY(int)
    ⊟ extended by advice
        UpdateSignaling.afterReturning(): change()
    int getX()
    int getY()
  ⊞ void moveBy(int, int)
⊟ Line implements Shape
    Point p1
    Point p2
    void setP1(Point)
    void setP2(Point)
    Point getP1()
    Point getP2()
  ⊟ void moveBy(int, int)
    ⊟ extended by advice
        UpdateSignaling.afterReturning(): change()
⊟ UpdateSignaling
```

void setX(int)

Sets the horizontal position to a given argument.

Extended if:
the target's apparent type is Point

pointcut change() :
The pointcut that captures the call of all setter methods and the call of `moveBy(int, int)` methods in the subclasses of Shape.

An after advice signals the `Display` to update whenever a shape changes.

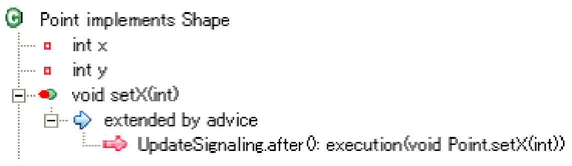
Figure 3.5. AspectScope

cutting structures by showing how aspects affect the module interfaces in the program. Like AJDT, the tool performs a global analysis of the deployment configuration of aspects but it presents the result of the analysis from the viewpoint of how the module interfaces of classes are extended by aspects. In other words, our tool projects AOP structure onto normal OOP (Object-Oriented Programming) structure so that developers can see crosscutting structures through their familiar OOP view. For example, the tool does not distinguish the call pointcut and the execution pointcut because the influence of these pointcuts on module interfaces is equivalent. It abstracts away from language-level differences between call and execution.

AspectScope consists of two panes: one for showing an outline view of a given class and the other for presenting javadoc comments describing the behavior of a selected method or field. These two panes reflect the extensions by woven aspects. See Figure 3.5.

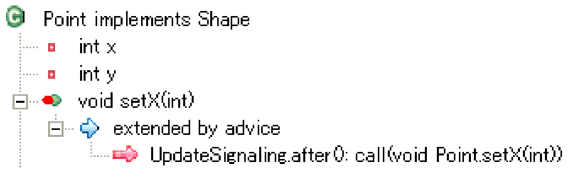
3.2.1 Outline view

The outline view by AspectScope lists methods and fields declared in a given class. It also shows whether or not the behavior of each method or field is extended by an aspect.



```
Point implements Shape
  int x
  int y
  void setX(int)
    extended by advice
      UpdateSignaling.after(): execution(void Point.setX(int))
```

Figure 3.6. The outline view presents the effect of the execution pointcut.



```
Point implements Shape
  int x
  int y
  void setX(int)
    extended by advice
      UpdateSignaling.after(): call(void Point.setX(int))
```

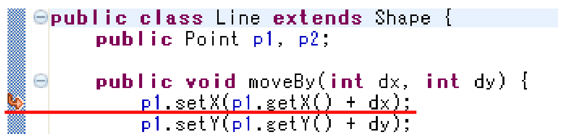
Figure 3.7. The outline view presents the effect of the call pointcut.

The execution and call pointcuts:

If an UpdateSignaling aspect includes an after advice associated with a pointcut execution(void Point.setX(int)), then the outline view indicates that the setX method in the Point class is extended by the after advice in the UpdateSignaling aspect (Figure 3.6).

Note that even if the pointcut that the after advice is associated with is not execution but call, for example, call(void Point.setX(int)), then the outline view shown does not change except the description of the pointcut (Figure 3.7). AspectScope abstracts away from differences between call and execution because module interfaces affected by aspects are interesting concerns. AspectScope considers that the advice associated with either pointcut extends the behavior of the *callee-side* method. In AspectJ, both pointcuts select method calls. However, the join points (or join point shadow [42]) selected by a call pointcut are method-call expressions at the *caller* side while the join points selected by an execution pointcut are the bodies of the specified methods at the *callee* (or *target*) side. Hence, for example, the advice associated with a call pointcut can obtain a reference to not only the target object but also the caller object. On the other hand, the advice associated with an execution pointcut cannot obtain such a reference.

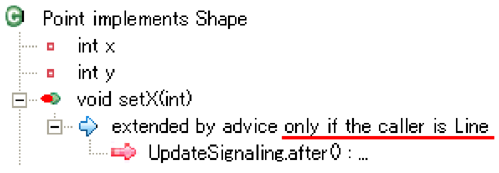
Despite this difference, AspectScope uses the outline view of the *callee* side to indicate the extension by the call pointcut. Since the goal is to display the module interfaces affected by aspects, AspectScope must project the extension to a module interface, which is the outline view of the callee side in OOP. On the other hand, AJDT reflects this difference. Figure 3.8 illustrates AJDT's visualization of the call pointcut shown above. An arrow icon indicates that the call to setX within the moveBy method is one of the selected join points. Note that the source code in this figure is of the caller-side method moveBy. AJDT does not show any information in the source code of the setX method, which is at the callee side.



```
public class Line extends Shape {
    public Point p1, p2;

    public void moveBy(int dx, int dy) {
        p1.setX(p1.getX() + dx);
        p1.setY(p1.getY() + dy);
```

Figure 3.8. AJDT indicates the effect of the call pointcut (the red underline was drawn by the authors).



```
Point implements Shape
    int x
    int y
    void setX(int)
        extended by advice only if the caller is Line
            UpdateSignaling.after() : ...
```

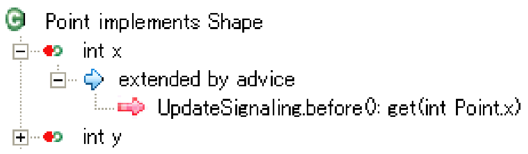
Figure 3.9. A conditional extension by the within pointcut (the red underline was drawn by the authors)

The within and cflow pointcuts:

The within, withincode, cflow, and cflowbelow pointcuts select join points within a specified region. For example, the within pointcut selects only the join points included in the specified class. call(void *.setX(int)) && within(Line) selects method calls from the Line class to setX declared in any class. The selected join points are method-call expressions contained in the body of a method in the Line class. The within pointcut restricts the *caller* methods.

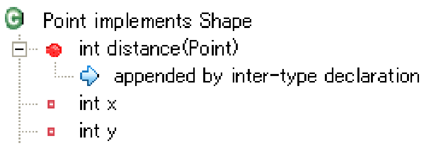
If the call pointcut is combined with the within pointcut, AspectScope interprets that the associated advice conditionally extends the behavior of the *callee* method. This is also true for the combination of call and cflow, set and within, and so forth. For example, if an UpdateSignaling aspect includes an after advice associated with a pointcut call(void Point.setX(int)) && within(Line), then the outline view indicates that the setX method in the Point class is *conditionally* extended by the after advice (Figure 3.9). Since the pointcut includes within(Line), the outline view shows that the behavior of setX is conditionally "extended by advice only if the caller is Line". The developers can see that the behavior of setX remains original if it is called from other classes than Line. If the combined pointcut is cflow, the outline view will show something like "extended if the thread is in the control flow of ..."

This visualization is different from AJDT. In AJDT, the influence of the within pointcut is equal between call and execution pointcuts. The within pointcut simply restricts the places indicated by arrow icons. In the case of the above pointcut, AJDT displays arrow icons only at the setX method calls that appear in the declaration of the Line class. AJDT does not show any information in the source code of the callee-side method setX.



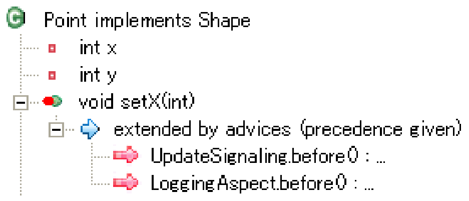
```
Point implements Shape
  int x
    extended by advice
      UpdateSignaling.before(): get(int Point.x)
  int y
```

Figure 3.10. There is a before advice associated with the get pointcut.



```
Point implements Shape
  int distance(Point)
    appended by inter-type declaration
  int x
  int y
```

Figure 3.11. An intertype declaration of the distance method



```
Point implements Shape
  int x
  int y
  void setX(int)
    extended by advices (precedence given)
      UpdateSignaling.before() : ...
      LoggingAspect.before() : ...
```

Figure 3.12. Two advices extend the setX method.

Other features:

The presentation of the get and set pointcuts in the outline view is similar to the call pointcut. In AspectJ, the join points selected by get and set pointcuts are field-access expressions at the accessor side (*i.e.* the caller side). Hence, AJDT shows an arrow icon at the line where the field is accessed. However, AspectScope interprets that an advice associated with a get or set pointcut extends the behavior of the target field. Figure 3.10 is an outline view presented by AspectScope. It illustrates the influence of an UpdateSignaling aspect that contains a before advice associated with a pointcut get(int Point.x). Note that an arrow icon is shown below the x field in the Point class (*i.e.* at the target side) because the advice extends the behavior of the x field.

An aspect may include an intertype declaration. The methods and the fields appended by intertype declarations are also shown in the outline view. For example, Figure 3.11 indicates that an intertype declaration appends the distance method to the Point class.

If more than one advice extends a method or a field in an existing class, the outline view lists all the advices. If precedence rules are given by declare precedence, the multiple advice bodies extending the same method or field are listed in the execution order satisfying the given precedence rules (Figure 3.12). On the other hand, AJDT does not show the execution order of multiple advices.

Limitation:

AspectScope does not support all the language constructs of AspectJ. For example, AspectScope does not show any information of advice if the pointcut associated with that advice is the handler pointcut. The handler pointcut selects join points that represent the time when an exception is caught by a catch clause. An advice associated with this pointcut cannot be regarded as an extension to a method but it should be regarded as an extension to a try-catch statement. It is a more fine-grained extension and thus the visualization by AJDT would be more appropriate than AspectScope. Otherwise, it might be regarded as an extension to the behavior of an exception class, or a subclass of Throwable, because the advice modifies how instances of a particular exception class is handled. This is an open question.

3.2.2 Refactoring revisited

In Section 3.1, we presented an example of the figure editor. See Figure 3.1. When refactoring, the developer who uses AJDT could not see that a call to the moveBy method in the Line class is also advised. To know this fact, she has to see the source code of UpdateSignaling aspect or use the Call Hierarchy view of Eclipse IDE to visit all the caller sites to moveBy, which is a manual whole-program analysis.

AspectScope provides better help than AJDT in this scenario of refactoring. When an experienced developer does this refactoring, what does she do first? Before she starts editing the program, she will first check the specifications of the moveBy method in Line, which she is going to modify for refactoring. She will look at the outline view shown by AspectScope to confirm whether or not the specifications of it is extended. Then she will also check the specifications of the setX and setY methods in Point because she will use them when modifying the body of moveBy. Again, she will look at the outline view shown by AspectScope. Note that AspectScope also shows the javadoc-style description of the specifications of a selected method such as setX (Figure 3.5). It also help the developer understand a crosscutting structure in the program. We will mention details of the javadoc-style description in the next subsection.

Since the views shown by AspectScope tell her that the methods are extended by an aspect, she will soon understand that the naive implementation causes redundant display updates (Figure 3.2). She will also understand that a call to the moveBy method is advised and thus calling moveBy causes five display updates in total. Therefore, before editing the source code of the moveBy method, she can know that she must also modify the change pointcut in the UpdateSignaling aspect. Note that AJDT does not show her that the setX and setY methods are advised until she actually edits the source code of the moveBy method. After she writes a method-call expression to setX in the body of moveBy, AJDT marks the expression with an arrow icon that indicates the setX method is advised.

AspectScope displays the influence of an aspect in the outline view of the *callee-side* classes even if the aspect selects *caller-side* join points by the call pointcut and so on. This is a simple idea but it helps developer's modular reasoning. In typical OOP, the callee-side outline view corresponds to a module interface. The visualization by AspectScope is to project AOP structure onto module interfaces of OOP, which developers are familiar with. This is why the influence of an aspect is displayed in the outline view of callee-side classes.

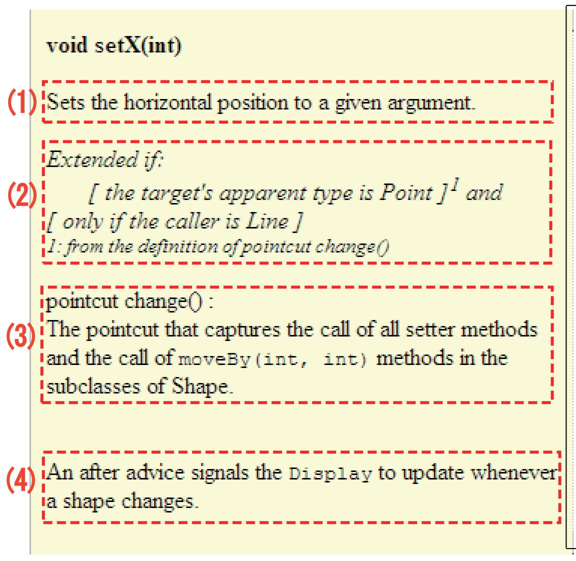
Some readers might think that looking at the outline views of the setX and setY methods in our refactoring scenario is a sort of manual whole-program analysis. This is not true because the outline views are part of module interfaces. If developers are looking at only the implementation of a local module and the interfaces of other modules, then it can be said that they are doing local reasoning.

3.2.3 Javadoc pane

AspectScope provides not only the outline view but also the javadoc pane. The right pane of the AspectScope is the javadoc pane. It displays the javadoc comments of a selected member, such as a method and a field, in the outline view. The displayed javadoc comments are extracted not only from the source code of the selected member but also from aspects extending the member. Developers can read the comments to see details of the extension by the aspect, in other words, how the aspect affects the module interface.

The contents:

Figure 3.13 is a screen snapshot of the javadoc pane. It is displaying the javadoc comments of the setX method in the Point class. We assume that the pointcut and advice listed in Figure 3.14 were woven with the Point class. The displayed javadoc comments consist of four parts.



void setX(int)

(1) Sets the horizontal position to a given argument.

(2) *Extended if:*
[the target's apparent type is Point]^1 and
[only if the caller is Line]
1: from the definition of pointcut change()

(3) pointcut change() :
The pointcut that captures the call of all setter methods and the call of `moveBy(int, int)` methods in the subclasses of Shape.

(4) An after advice signals the `Display` to update whenever a shape changes.

Figure 3.13. The javadoc pane for the setX method (the red dotted lines and text were drawn by the authors)

```
/**
  * The pointcut that captures the call of all
  * setter methods and the call of
  * ⟨code⟩moveBy(int, int)⟨/code⟩ methods
  * in the subclasses of Shape.
  */
pointcut move(): call(void Shape+.set*(int))
                  || call(void Shape+.moveBy(..));

/**
  * An after advice signals the
  * ⟨code⟩Display⟨/code⟩ to update whenever
  * a shape changes.
  */
after(): move() && within(Line) {
      Display.update();
}
```

Figure 3.14. The definition of an advice and a pointcut

First, the text in (1) is constructed from the javadoc comments in the source code of the setX method. They describe the original behavior of the method. If any aspect is not deployed to extend the behavior of the setX method, the javadoc pane of AspectScope displays only this text.

The text in (2) to (4) is constructed from the source code of the aspect. If there are multiple aspects woven, the text is constructed for each aspect. The text in (2) describes that the behavior of the setX method is extended by an aspect. If the extension is conditional, the text in (2) also describes that condition. It is an English translation of the pointcut associated with the advice that extends the setX method. Note that it is not a naive translation of the pointcut expression, which is move() && within(Line). AspectScope expands a named pointcut such as move and removes unnecessary pointcuts. For example, call(void Shape+.moveBy(..)) is unnecessary because we are now interested only in the setX method; this pointcut never matches. call(void Shape+.set*(int)) is also redundant for the same reason. Since the method is setX in Point, AspectScope first expands wild-cards and displays an English translation of call(void Point.setX(int)). According to AspectJ's specification, the call pointcut selects join points by using the apparent type of a target object. Thus, AspectScope displays that the behavior of setX is extended only if the apparent type of the target is Point.

Recall that the execution pointcut uses the actual type of a target object. Thus, if the call pointcuts in Figure 3.14 were replaced with the execution pointcuts, then the text in (2) would not include the text related to the execution pointcut. AspectScope would never display "if the actual type is Point" because this phrase is redundant for describing when an advice extends the behavior of the setX method. When the setX method in Point is executed, the actual type of the target object must be Point! If there is no other pointcut remaining after unnecessary pointcuts are removed, AspectScope simply displays "Extended always" instead of "Extended if..." For example, if the after advice in Figure 3.14 were the following:

```
after():  execution(void Shape+.set*(int)) {
  Display.update();
}
```

then the text in (2) would be only "Extended always" because the behavior of the setX method is unconditionally extended by the after advice.

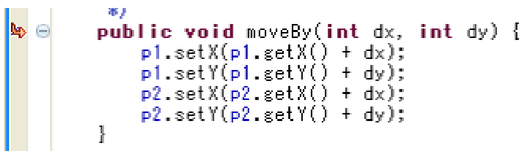
The text in (3) is constructed from the javadoc comments of the named pointcuts related to the setX method, such as the move pointcut. It is shown here for giving additional information on the condition of the extension by an aspect. Finally, the text in (4) is extracted from the source code of an advice that extends the behavior of the setX method. It describes details of that extension.

Wild cards:

When AspectScope shows a translation of pointcut in the javadoc view, it expands wild cards. For example, the wild cards in call(void Shape+.set*(..)) are expanded when AspectScope shows the javadoc comments for the setX method in Point. The result is call(void Point.setX(int)).

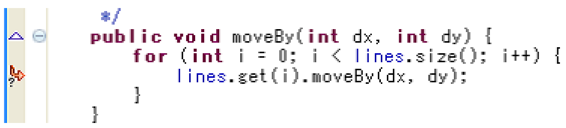
However, not expanding wild cards in a pointcut might be convenient, for example, when an aspect is homogeneous and implements a non-functional concern [1] such as access authentication. Developers might want to see the original pointcut containing wild cards. AspectScope always expands wild cards because it was designed for showing the module interface of each class

[1]A non-functional concern is a concern independent of the application logic. Thus it is often commonly used among different applications.



```
*/
public void moveBy(int dx, int dy) {
    p1.setX(p1.getX() + dx);
    p1.setY(p1.getY() + dy);
    p2.setX(p2.getX() + dx);
    p2.setY(p2.getY() + dy);
}
```

Figure 3.15. AJDT only shows that moveBy is advised.



```
*/
public void moveBy(int dx, int dy) {
    for (int i = 0; i < lines.size(); i++) {
        lines.get(i).moveBy(dx, dy);
    }
}
```

Figure 3.16. The arrow icon indicating a join point shadow of cflowbelow

under the deployment of aspects. It shows the javadoc comments for explaining how the behavior of a selected method or field is extended. If an aspect is homogeneous and a single advice body extends the behavior of multiple classes, this fact should be written in javadoc comments of that advice or its pointcut. Developers will be able to see the existence of the homogeneous aspect when they read the javadoc comments of one of the target method of that aspect through AspectScope.

3.3 Examples

In Section 3.2.2, we have already shown an example of AspectJ programming with AspectScope. We below show a few other examples.

3.3.1 Using the execution pointcut

AspectScope still provides a different visualization from AJDT's one even if the change pointcut in UpdateSignaling shown in Figure 3.1 is replaced with the following:

```
pointcut change():
   execution(void Point.setX(int))
   || execution(void Point.setY(int))
   || execution(void Shape+.moveBy(int,int));
```

Here, the execution pointcut is substituted for the call pointcut.

Since AspectScope deals with call and execution alike except translations, it shows the same outline view and the same javadoc comments as in the previous Section 3.2.2. Developers can still do modular reasoning.

On the other hand, AJDT only tells developers that the moveBy method is advised (Figure 3.15). They cannot see that the setX and setY are also advised. They must browse the source code of these methods. Some readers might think that browsing the source code of the methods is natural if the developers want to use them in the moveBy method. However, browsing the source code is not equal to looking at the module interfaces of the methods. It is rather looking at the internal implementation of a module and hence it is breaking the principle of information hiding. Of course, if appropriate javadoc comments are not provided by the developer, the users of AspectScope might also have to browse the source code of setX and setY. There is no serious difference between AJDT and AspectScope in that case.

3.3.2 Denotation of cflowbelow pointcut

To fix the problem of redundant display updates in Section 3.1, the after advice in the UpdateSignaling aspect must be updated to be this:

```
after() returning:  change() && !cflowbelow(change()) {
   Display.update();
}
```

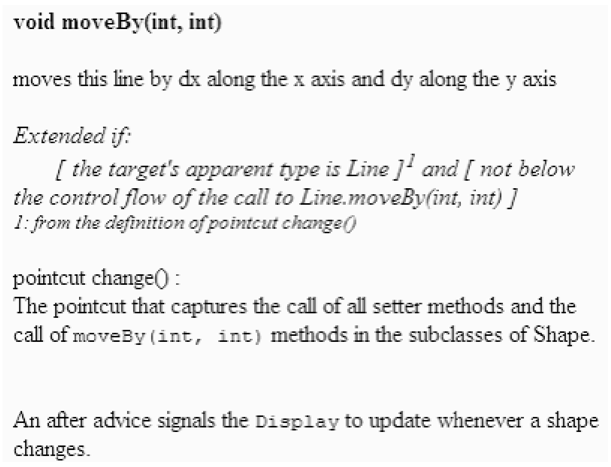
AspectScope presents better representation after this update than AJDT.

As illustrated in Figure 3.16, AJDT displays an arrow indicating a join point shadow at the line where the moveBy method is called. However, this arrow icon does not show any extra information. Developers must click this icon to jump to the source code of the advice woven there. If they do not click, they cannot see the join points are selected by cflowbelow. On the other hand, AspectScope shows this fact within the javadoc comments of the moveBy method (Figure 3.17). The javadoc pane mentions that the moveBy method is extended "only if the apparent type is Line and not below the control flow of the call to Line.moveBy(int, int)". This fact is also shown in the outline view. Developers will be able to naturally see the exact effects of the UpdateSignaling aspect.

3.3.3 Defining a new class implementing Shape

The final example is to define a new class implementing Shape for the figure editor. Let the name of the new class be Circle. The Shape interface is defined in Figure 3.1.

A developer who will define the Circle class would want to know that the moveBy method in Circle is extended by the UpdateSignaling aspect. However,



void moveBy(int, int)

moves this line by dx along the x axis and dy along the y axis

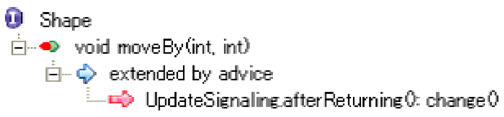
Extended if:

[the target's apparent type is Line]^1 and [not below the control flow of the call to Line.moveBy(int, int)]
1: from the definition of pointcut change()

pointcut change() :
The pointcut that captures the call of all setter methods and the call of `moveBy(int, int)` methods in the subclasses of Shape.

An after advice signals the `Display` to update whenever a shape changes.

Figure 3.17. The javadoc pane mentions cflowbelow.



Shape
void moveBy(int, int)
extended by advice
UpdateSignaling.afterReturning(): change()

Figure 3.18. The notification of moveBy method

AJDT does not tell her this fact until she defines the Circle class and writes client code. This is an example of undesirable obliviousness. She has to start writing the Circle class without knowing the extension by the aspect.

If the developer is an experienced engineer, she would first think that she should read the specifications of Shape. This is natural because she is going to define a class implementing the Shape interface. AspectScope helps such an experienced engineer. If she looks at the outline view that AspectScope shows for the Shape interface, she will notice that the moveBy method will be extended by the UpdateSignaling aspect (Figure 3.18). She can first know the extension by the aspect and then start writing the Circle class.

3.4 Preliminary evaluation

To evaluate the usefulness of AspectScope, we used it for browsing the source program of ActiveAspect [31], which is a programming tool for AspectJ written by the third party. The program is written in AspectJ and it consists of 88 classes (10,683 lines) and 19 aspects (2,477 lines).[2]

[2]Since the original program has a few bugs, we did this study after fixing the bugs.

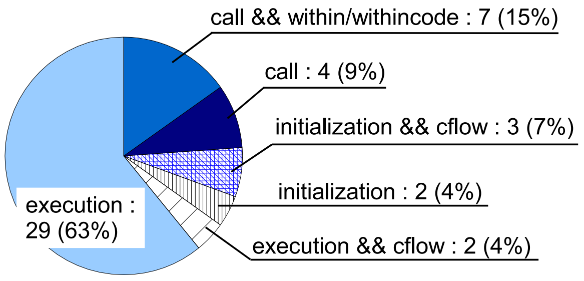


Figure 3.19. The analysis of the source program of ActiveAspect

3.4.1 The frequency of simple execution pointcuts

For the execution pointcut, the outline view of AspectScope is almost equal to the visualization by AJDT unless the execution pointcut is used with other pointcuts like cflow. Thus, if most pointcuts used in typical AspectJ programs are simple execution, the benefit of using AspectScope is relatively small.

Figure 3.19 shows the number of each pointcut designator used in the program of ActiveAspect. 67% of all the pointcuts are simple execution or initialization, which are expressed in the same way between AspectScope and AJDT. The join points selected by initialization are (part of) the execution of a constructor. The rest of the pointcuts are visualized by AspectScope in a different way from AJDT. They are pointcuts including call, within, withincode, or cflow.

3.4.2 The callee-side extension

When AspectScope visualizes call pointcuts, it interprets them as callee-side extensions although they select join points at the caller side (*i.e.* the client side). This is because it displays the effects of the pointcuts as changes of module interfaces, which are the outline view of the callee-side. We reviewed the program of ActiveAspect to examine whether or not this interpretation by AspectScope is acceptable for each pointcut.

A call pointcut combined with no other pointcut such as within selects method calls from any client site. Thus, an advice executed at these join points can be regarded as either a callee-side extension or a caller-side extension. There is no serious difference between the two interpretations; it is a natural interpretation that the advice is a callee-side extension.

Pointcut	Comments by the authors
execution(* SelectionOperator.apply()) && target(selector) && !cflow(execution(* MemberExpander.*(..)))	apply sets a flag of its target object except during the execution of a method in MemberExpander.
execution(* MemberEditPart.mousePressed(MouseEvent)) && target(ePart) && args(me) && !cflow(adviceexecution())	mousePressed performs an extra action depending on the state of the aspect instance. !cflow(..) is for avoiding infinite recursion.
call(ModelRelationship.new(..)) && !within(ModelRelationship)	The constructor displays an error message if it is not called from a singleton factory method.
(call(* ModelElement.addSourceRelationship(..)) || call(* ModelElement.addTargetRelationship(..))) && !within(ModelRelationship)	The methods add.. display a warning message if they are called from classes except ModelRelationship.
(call(* ModelElement.getModelCopy(..)) || call(ModelElement.new(..))) && !within(ProgramModel+) && !within(ModelElement+)	getModelCopy and the constructor display a warning message if they are called from classes except the specific classes.
call(* ModelRelationship.setHidden(boolean)) && args(boolean) && target(ModelRelationship) && !within(StickyRels)	!within(StickyRels) avoids the infinite recursive execution of this advice in StickyRels.
call(* IDrawableEntity.setLocation(..)) && target(ModelElement) && withincode(* ClassifierEditPart.createFigure(..))	setLocation performs an extra action if it is called from the createFigure method in ClassifierEditPart.
initialization(AggregateRelationship+.new(..)) && target(ModelRelationship) && cflow(execution(void AbstractMembersRule.apply()))	The constructor sets a flag of the created object if it is called during the execution of the apply method.
initialization(AggregateRelationship+.new(..)) && target(ModelRelationship) && cflow(execution(void AbstractRelationsRule.apply()))	the same as above except apply is a method in not AbstractMembersRule but AbstractRelationsRule. This pointcut is used by two advices, which set a different flag.

Table 3.1. All pointcuts declared in the source code of ActiveAspect

A call pointcut combined with a target pointcut also selects method calls from any client site. The target pointcut restricts the actual type of the target object. An advice associated with such a call pointcut can be also regarded as a callee-side extension. An example we found in the program of ActiveAspect was the following:

```
after(Object modelObj, AbstractEditPart editPart):
   call(void EditPart+.setModel(Object)) && args(modelObj)
                                         && target(editPart) {
   ((IDrawableEntity)modelObj).setEditPart(editPart);
}
```

This advice makes a reverse link from the argument to the target object when the setModel method is called on an AbstractEditPart object. AbstractEditPart is a subclass of EditPart. It is natural interpretation that this advice extends the callee-side setModel method.

Interesting pointcuts with respect to interpretation are call pointcuts combined with within, withincode, or cflow. The execution and initialization pointcuts with cflow are also interesting. Because within and cflow restrict caller-side contexts, we thought that it might be less natural to interpret the advices combined with such pointcuts as callee-side extensions. However, as we listed in Table 3.1, we could not find any advices that *must* be interpreted as caller-side extensions. For example, the following code is one of the advices that it is least natural to interpret as callee-side extensions:

```
after(ModelElement elt):
    call(* IDrawableEntity.setLocation(..))  && target(elt)
        && withincode(* ClassifierEditPart.createFigure(..))  {
    setLocation(elt);
}
```

This advice changes the behavior of the setLocation method only when it is called from the createFigure method. setLocation(elt) calls a method declared in the aspect including the above advice. It performs the dedicated action for playing the demo of the software. Since the behavior depends on the caller site, it is somewhat inappropriate to interpret this advice as a pure callee-side extension. However, this interpretation is still acceptable.

In summary, the interpretation by AspectScope was not a serious problem in our preliminary case study. However, the program of ActiveAspect does not include non-functional homogeneous aspects, such as logging and authentication, which may not fit the interpretation by AspectScope. Although Apel et al. also reported that such aspects are not frequently used [9][3], we need further study on this topic.

3.5 Summary

AspectScope performs a whole-program analysis of AspectJ programs and visualizes the result so that developers can understand their program behavior

[3]In [9], most aspects are implemented by mixins. They correspond to AspectJ's advices associated with the execution pointcut.

with local reasoning. It displays the module interfaces extended by aspects under current deployment.

A unique idea of AspectScope is to interpret an aspect as an extension to the callee-side (target-side) class even if the aspect includes the call pointcut. This enables expressing the effects of aspects through module interfaces. Developers thereby do AOP by using their OOP experiences of modular programming, in particular, modular extensions to classes by virtual classes [66], mixin-layers [80], nested inheritance [68, 69], and so on.

On the other hand, AspectScope is inappropriate for aspects that this interpretation does not fit although such aspects would be not many. For such aspects, we should switch tools to AJDT. A tracing aspect for debugging and a transaction aspect often fall into this category. Such aspects interpret join points as *events* that triggers the execution of advice code [37, 26, 6]. For example, they use a call pointcut for executing an advice in the middle of the method body of the caller. Such an advice is independent of the *callee* side and it is used only for extending the behavior of the *caller-side* (client-side) method. It should be regarded as a caller-side extension. Although those aspects are also significant applications of AOP, the influence of the aspects on module interfaces is difficult to express.

Chapter 4

CommentWeaver

To address problems that occur when API documentation is published, this chapter 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 chapter 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.

4.1 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.

4.1.1 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 4.1 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.

4.1.2 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 [48] or the tyranny of the dominant decomposition [72].

We below illustrate this crosscutting problem by showing a few examples taken from the Javassist library [21]. 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 4.1 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 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 4.1 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

```
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 4.1. Scattering text in the doc comments

```
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 4.2. Tangling text in the doc comment

because the users would not read the description of toBytecode when they write a user program that calls writeFile. They would not know that writeFiles internally call toBytecode. This fact is an implementation detail that should be hidden from the users according to the information hiding principle [73]. 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 4.2 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 4.2 is that the text "If defrost() will 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 [2] 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 4.3 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.

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.

4.2 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

```
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 4.3. Text that should belong to the doc comment in an aspect

@quote $((\textit{class-name}.)?\ \textit{member-name})\ (.\textit{export-name})?$

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

@export $(:\ \textit{export-name})?\ \{\ \textit{text}\ \}$

$\textit{text} :=$ @quote(..) $\textit{text} \mid \langle$normal text$\rangle\ \textit{text} \mid \langle$javadoc tags$\rangle \mid \phi$

@weave $(\textit{pc})\ \{\ \textit{text}\ \}$

$\textit{pc} :=$ call $(\textit{method-pattern}) \mid$ exec $(\textit{method-pattern}) \mid$
within $(\textit{class-}$ or $\textit{method-pattern}) \mid$
$\textit{pc}\ \&\&\ \textit{pc} \mid \textit{pc} \parallel \textit{pc} \mid !\ \textit{pc}$

In the doc comment of an advice, also
$\textit{pc} :=$ JP | JP_CALLER | JP_CALLEE

@liftup $\{\ \textit{text}\ \}$

Figure 4.4. Syntax of CommentWeaver tags

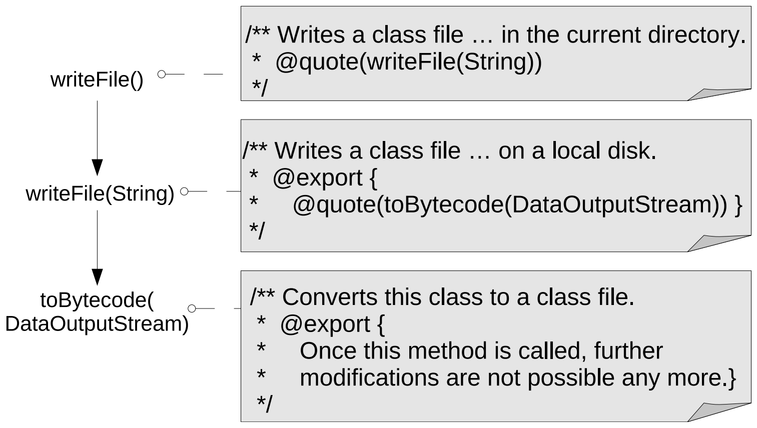

writeFile()
writeFile(String)
toBytecode(DataOutputStream)
/** Writes a class file … in the current directory.
* @quote(writeFile(String))
*/
/** Writes a class file … on a local disk.
* @export {
* @quote(toBytecode(DataOutputStream)) }
*/
/** Converts this class to a class file.
* @export {
* Once this method is called, further
* modifications are not possible any more.}
*/

Figure 4.5. @quote and @export tags in the CtClass class

documentation of several other methods, which are different from the methods that the doc comments are originally attached to. The appended 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 4.4.

4.2.1 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 4.1 of Section 4.1.2. Figure 4.5 illustrates the result of rewriting the program in Figure 4.1 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 write-File(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 4.5 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 {
```

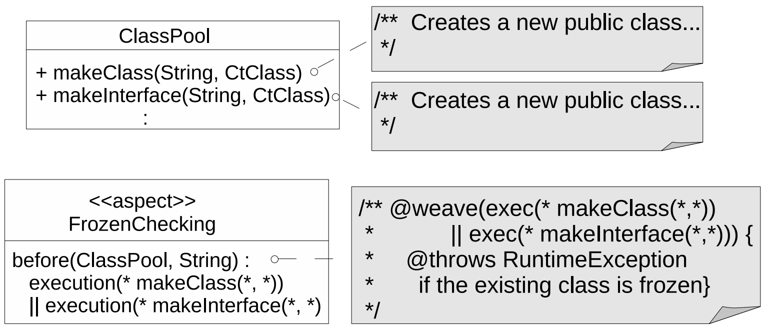

ClassPool
+ makeClass(String, CtClass)
+ makeInterface(String, CtClass)
:
/** Creates a new public class...
*/
/** Creates a new public class...
*/
<<aspect>>
FrozenChecking
before(ClassPool, String) :
execution(* makeClass(*, *))
|| execution(* makeInterface(*, *)
/** @weave(exec(* makeClass(*,*))
* || exec(* makeInterface(*,*))) {
* @throws RuntimeException
* if the existing class is frozen}
*/

Figure 4.6. A doc comment moved into an aspect

```
*   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.

4.2.2 Scattering text by aspect

Although aspects modularize crosscutting concerns for programming, they do not for doc comments as we mentioned in Section 4.1.2. 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 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 4.3. Figure 4.6 shows the result of the rewrite for CommentWeaver. The difference between Figure 4.3 and 4.6 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 4.6, 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 4.6, 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 4.6 can be simplified as the following:

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

Since CommentWeaver is a compile-time tool, the variable JP represents join point shadow [42] 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 4.6, 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.

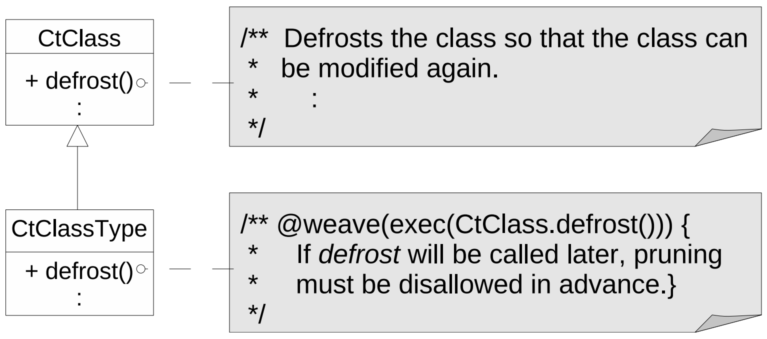
Developers might think the description of RuntimeException should be included also in the API documentation of the caller methods that call the makeClass and makeInterface in the ClassPool class. If so, the doc comments of the before advice could be modified into the following.

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

The variable JP_CALLER represents the caller methods.

4.2.3 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



```
/**  Defrosts the class so that the class can
 *   be modified again.
 *      :
 */
```

```
/** @weave(exec(CtClass.defrost())) {
 *     If defrost will be called later, pruning
 *     must be disallowed in advance.}
 */
```

Figure 4.7. A doc comment moved to a public super class

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 4.7 is the result of rewriting the program in Figure 4.2 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 4.7 can be replaced with "@liftup { If defrost will be ... }".

4.2.4 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.

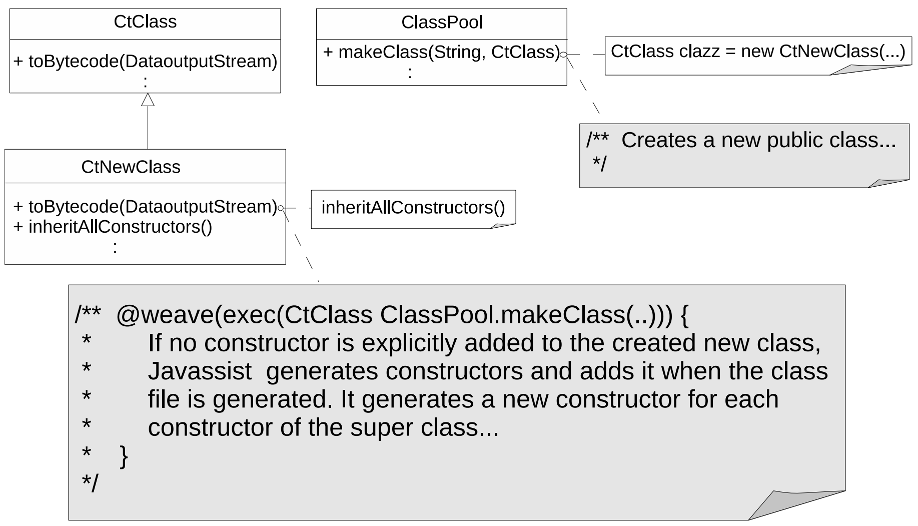

CtClass
+ toBytecode(DataoutputStream)
ClassPool
+ makeClass(String, CtClass)
CtClass clazz = new CtNewClass(...)
/** Creates a new public class...
*/
CtNewClass
+ toBytecode(DataoutputStream)
+ inheritAllConstructors()
inheritAllConstructors()
/** @weave(exec(CtClass ClassPool.makeClass(..))) {
* If no constructor is explicitly added to the created new class,
* Javassist generates constructors and adds it when the class
* file is generated. It generates a new constructor for each
* constructor of the super class...
* }
*/

Figure 4.8. A doc comment moved to a factory method

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) [4]. 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.

4.2.5 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:

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

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.

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

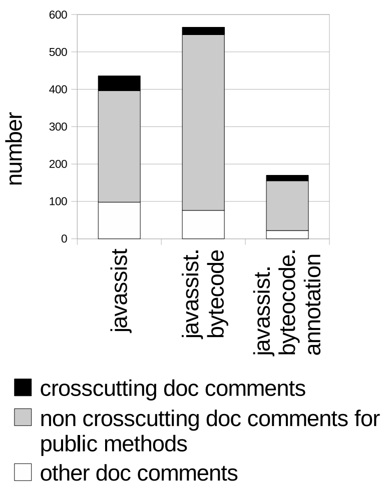
If m does not call m', then a compile error is reported.

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.

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



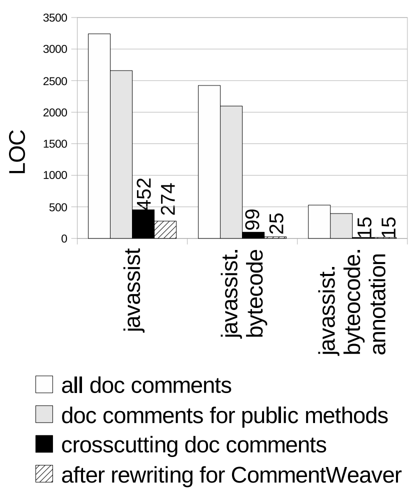


Figure 4.9. The doc comments for Javassist

$$pce(w)\ matches\ m\}$$

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.

4.3 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 4.1.2 can be rewritten to be more modular by CommentWeaver. This section discusses the applicability of CommentWeaver to existing class libraries.

4.3.1 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 4.9 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 4.9 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

[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.

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 4.1.

4.3.2 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 4.10 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 4.11 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.

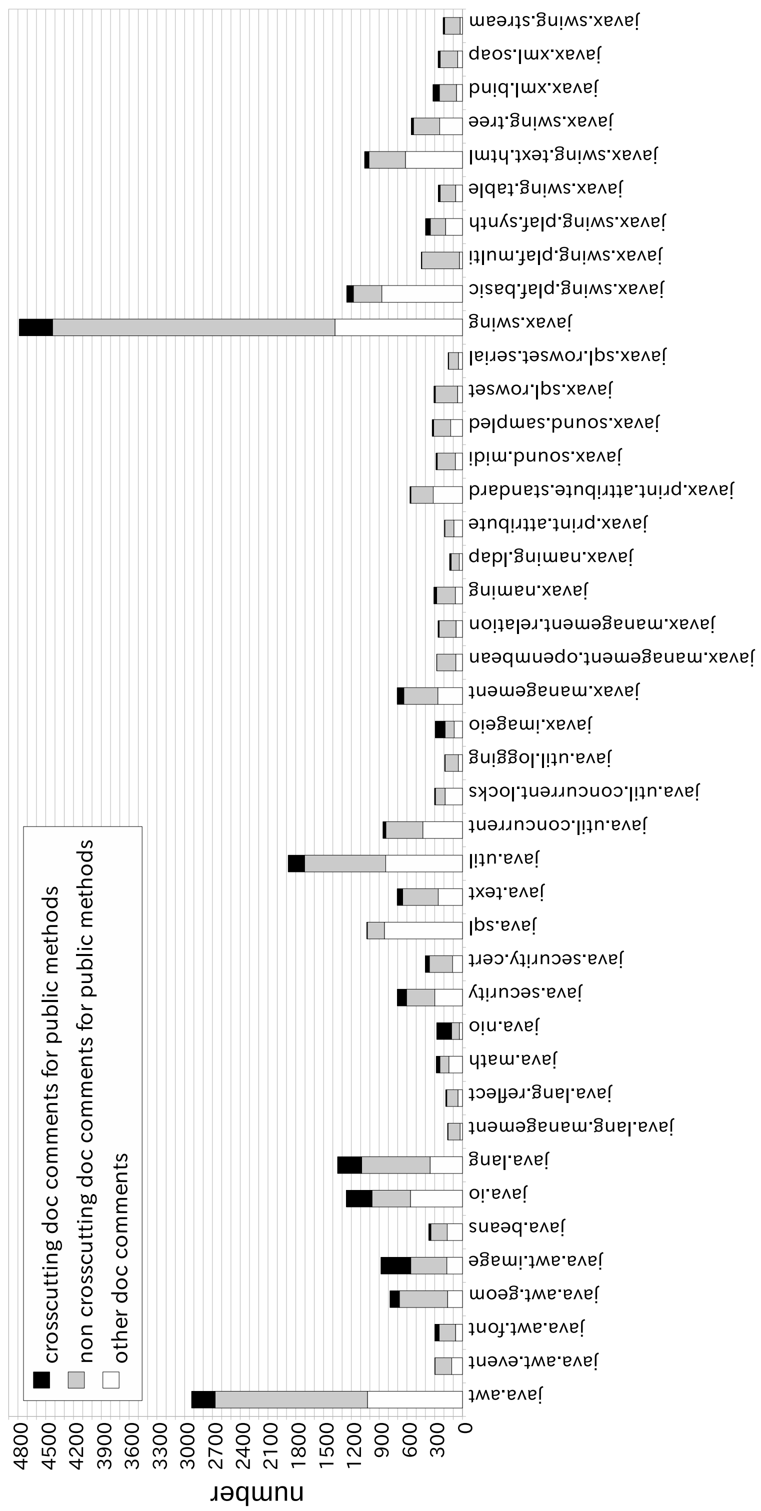


Figure 4.10. The crosscutting doc comments in Java 6

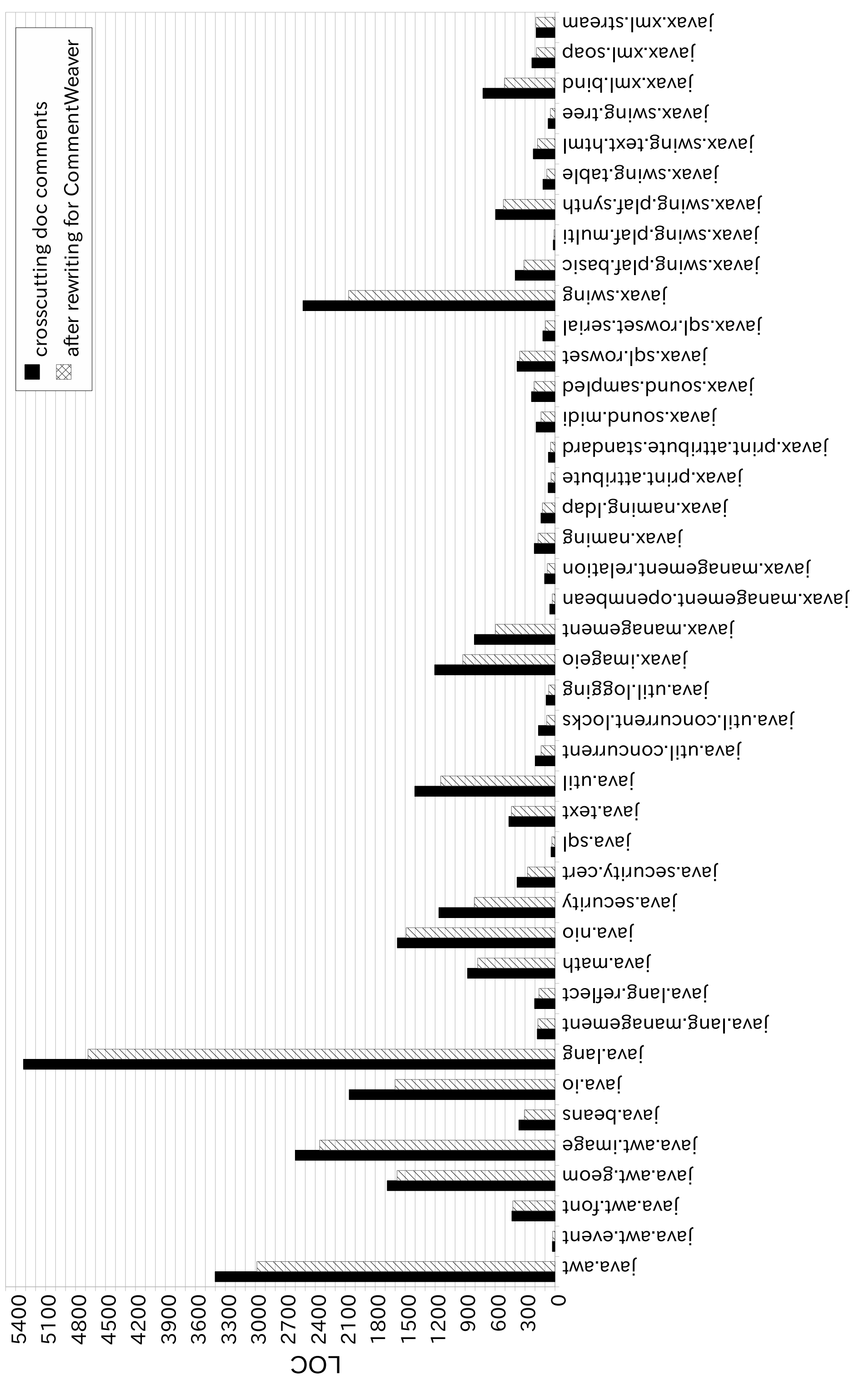


Figure 4.11. The crosscutting doc comments in Java 6

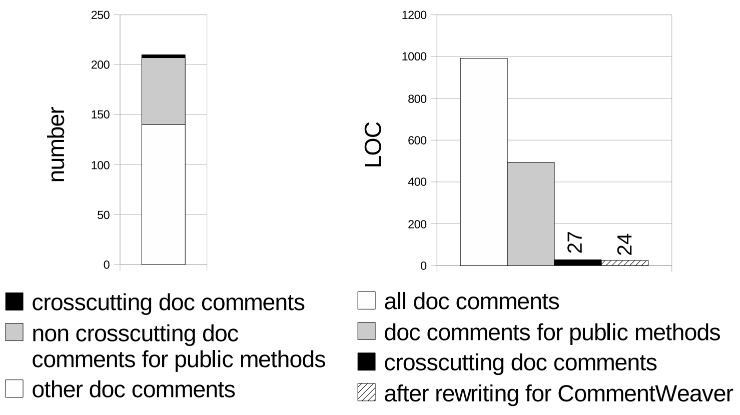


Figure 4.12. The doc comments for Eclipse (on average per package)

4.3.3 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 4.12 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 4.1. The tangling text in the doc comments for the CtClass class

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 4.2. The aspects implemented for Javassist

4.3.4 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 4.2 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 4.2 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 4.2 shows that the JP_CALLEE was necessary for several cases.

4.4 Summary

This chapter 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.

Chapter 5

Universal Aspect Oriented Programming

In order to demonstrate the feasibility of our argument, we present *Universal AOP* - a tool that provides modular views to developers to understand crosscutting concerns in programs. Developers do not need to learn new programming construct to understand crosscutting concerns in programs although the same effects that language based modularity provides is achievable.

The desire for improved modularity is one of the main issues of software development research and practice. Ever since the seminal work of Parnas [73], a major effort in Software Engineering research has been directed to develop various types of modularisation. In the area of programming languages design, this effort has resulted in a plethora of paradigms with their languages, such as procedural, functional, object-oriented, and more recently, aspect-oriented languages. However, as stated by Brooks [20], there is “no silver bullet” to this modularity problem with each of these methodologies having their own advantages and disadvantages, and each being suited to different circumstances. In order to gain the modularity benefits of the selected programming paradigm a representation has to be created using an appropriate programming language from within that paradigm. This creates an overhead with the subtleties and nuances of that language having to be learnt before it can be efficiently applied.

The two main benefits delivered by modularity are improved software reuse and maintainability. Reuse is achieved due to relative physical independence between modules, which can be subsequently used without change in different systems. Maintainability is achieved due to localisation of code, which can be viewed in one place, and changed locally, thus reducing the scope of the change. While the reuse property requires physical mobility of the code, maintainability can often be satisfied via on-demand localisation of the relevant code, irrespective of its physical location. To achieve these quality attributes, the selection of a suitable programming paradigm and language to begin development is of paramount importance. However, many studies such as [16] have found that maintenance activities incur the most significant costs during a system's life-time (up to 80%). Furthermore, some maintenance activities may be hindered due to the initial programming paradigm and language selected. For example, if a crosscutting concern needs to be maintained and the initial representation does not allow an appropriate view of this concern to be generated.

Our underlying idea is that documentation are necessary whether the development is large scale or not, and always written using natural language (NL), maintaining the same NL grammar and semantics, irrespective of what implementation technique or language is used for code. Moreover, the semantics of the code documentation is organically connected to the semantics and structure of the code. Thus, if modularization (and composition) is achievable in terms of natural language grammar and semantics, the actual structure of the implementation and its language are inconsequential. Universal AOP enables natural language text-based composition for crosscutting concerns in code documentation. Using this tool, we can define composition of crosscutting concerns using the NL documentation inlined in the code, yet, without direct reference to the code itself. Currently, documentation result can be viewed through HTML files like the API documentation.

Our solution involves utilizing attributes of natural language comments to generate alternative modularisation views to the one provided by the programming paradigm used to implement the system for the purpose of software maintenance. For example, a concern-oriented view can be generated from a procedural implementation or an aspect-oriented view can be generated from a purely object-oriented implementation. By using comments in this way, problems associated with applying a new programming paradigm such as: fragility, composition definition and composition comprehension can be minimised. The cost of learning a new language, and its adoption risk are also removed. In addition, a new type of modular view can be constructed using the existing implementation comments for existing software, delivering the maintainability properties, irrespective of the underlying modularity of the software.

5.1 Fragile document composition against changes

The weaving mechanism of CommentWeaver is syntax-based as well as current AOP languages are. As a result, they come with the well documented problems of pointcut fragility [56] and difficulty of composition definition and comprehension although there are plenty of researches to avoid this fragility problem [7, 34, 35, 36, 51, 75, 79, 8, 82, 41]. The issue of pointcut fragility arises due to direct referring of the code structure and name used in the pointcut specification. At any time when code structure or naming change, the pointcut specification becomes invalid. Similarly, due to direct referencing to the code structure and naming used in the pointuct specification, the developer is expected to be aware about all names and locations that are to be included into the pointcut, making composition definition difficult.

5.1.1 Refactoring in Javassist

Software is often updated through several versions. Although a specification might be added or changed in some versions, most frequent update will be refactorings of its implementation. As mentioned in the previous chapter, CommentWeaver will provide a way to compose documentation by specifying concrete program structures to produce API documentation. It is reasonable that CommentWeaver uses program structures because API documentation is one of the final products of libraries or frameworks. For example, as shown in Figure 5.1.1, five writeFile methods were defined in version 2.6. In version 3.1, refactorings were carried out and the signatures of the writeFiles and caller-callee relations were also changed. The writeFile(String) method in the ClassPool class in version 2.6 was moved into the CtClass class and the String type parameter was removed in version 3.1. Also, the writeFile(Stirng, String) method in the ClassPool class was moved into the CtClass class and changed to have one String type parameter. Although three other writeFile methods were defined and they invoked the toBytecode method in the CtClass class (in)directly in version 2.6, all these methods were removed and the writeFile(String) in the CtClass invoked the toBytecode(DataOutputStream)

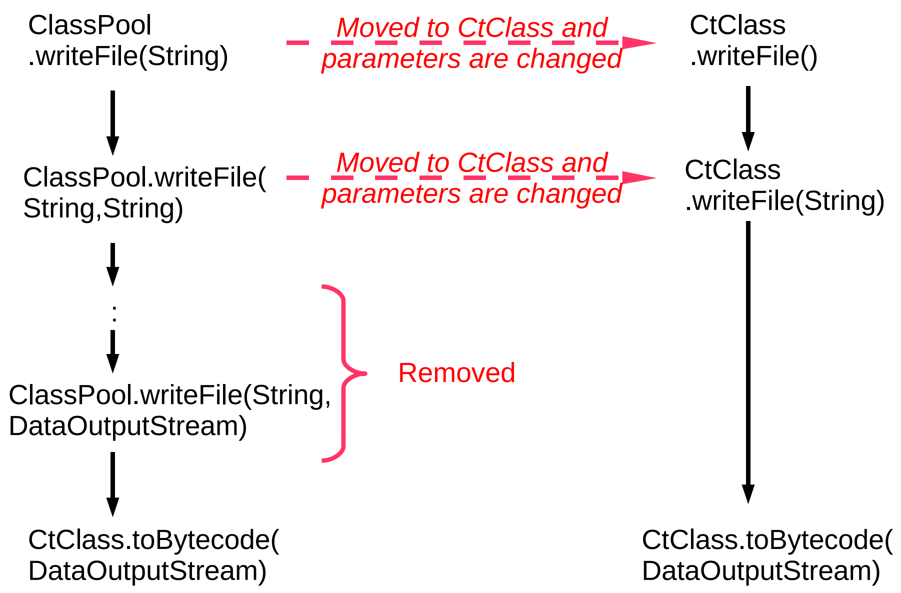


Figure 5.1. Refactoring in Javassist between version 2.6 to version 3.1

method directly in version 3.1. The toBytecode(DataOutputStream) method was also changed as *private* to *public*.
Since the text starting with "Once this method is called" that is written in the toBytecode method is necessary among the API documentation of all the method shown in Figure 5.1.1, CommentWeaver tags are needed to only modularize the text in the doc comment of the toBytecode method. In CommentWeaver, the @quote is used to refer to text through caller-callee relations. Therefore, the doc comment of the writeFile(String) method includes the @quote that refers to the concrete method name of another callee writeFile method as shown below. The doc comment of the writeFile(String,String) method also have to use @quote tag to further refer to the text.

```
/**
 * Writes a class file specified with <tt>classname</tt>
 * in the current directory.
 * @quote(writeFile(String,String))
 */
 public void writeFile(String classname) {
    writeFile(classname, ".");
 }

/**
```

```
 * Writes a class file specified with <code>classname</code>
 * on a local disk.
 * @export {
 *   @quote(writeFile(String,String,boolean))
 * }
 *      :
 */
 public void writeFile(String classname, String directoryName) { ... }
```

However, in version 3.1, the definitions of the @quote tags have to be changed as follows along with the refactoring. This will a heavy task when all these changes of CommentWeaver tags have to be done in each refactoring.

```
/**
 * Writes a class file specified with <tt>classname</tt>
 * in the current directory.
 * @quote(writeFile(String))
 */
 public void writeFile() {
    writeFile(".");
 }

/**
 * Writes a class file specified with <code>classname</code>
 * on a local disk.
 * @export {
 *   @quote(toBytecode(DataOutputStream))
 * }
 *      :
 */
 public void writeFile(String directoryName) { ... }
```

A naive solution

One of the solution to avoid the fragility of pointcuts in CommentWeaver is to use AspectJ. The @weave tag is available when text should be pushed into other doc comments from a doc comment. Since AspectJ aspects are implicitly invoked from classes and there is no description of invocation of aspects in classes, the @weave tag is useful because it does not need to be written anything in classes. As shown below, the argument of the @weave tag can take the JP to specify correspondent methods that are selected by an AspectJ pointcut. The JP_CALLER is used to specify the caller methods to the selected methods by an AspectJ pointcut. In this example, the JP specifies the writeFile(String) method because the AspectJ pointcut selects it.

The JP_CALLER specifies the writeFile method because it is a caller method to the writeFile(String).

```
public abstract class CtClass {
 /**
  * Writes a class file specified with <tt>classname</tt>
  * in the current directory.
  */
  public void writeFile() {
     writeFile(".");
  }

 /**
  * Writes a class file specified with <code>classname</code>
  * on a local disk.
  *      :
  */
  public void writeFile(String directoryName) { ... }
}

aspect BytecodeDescribing {

 /**
  * @weave(JP || JP_CALLER) {
  *   Once this method is called, further modification is not
  *   possible any more.
  * }
  */
  after() : execution(void CtClass.writeFile(String)) { ... }
}
```

However, the problem is that the former implementation that uses only Java are already well modularized and there might not be necessary to use AspectJ. Besides, to learn how to use AspectJ is not a simple task.

5.2 Modularization revisited

Two major benefits of good modularity are improved software reuse and maintainability. Reuse[1] is achieved through physical independence between

[1]The proposed approach does not directly support code reuse, as the generated views do not provide physical relocation of contained code into new modules. Yet, these views can inform the choice of other programming languages for potential code re-factoring and subsequent reuse.

modules. Such modules should be subsequently usable in other systems without change. Maintainability is achieved through localisation of code. Such code can be viewed and changed in one place, thus reducing the scope of the change. In this paper, we propose support for maintainability via views for virtual localisation of code. Since maintenance accounts for up to 80% [16] of system life-time costs, we consider supporting it a priority.

5.2.1 Programming Languages for Modularity

As described above, in programmes code modularity is achieved through selection of a programming abstraction and language suited for a given problem. For example, OOP came about when abstractions that reflect the real, object-based world were perceived necessary. More recently, AOP arose when the need for the modular representation of crosscutting concerns was recognised.

There have been a number of AOP languages so far but those languages provide special language constructs. Developers have to pay a not-small amount of costs to learn them. In fact, any AOP languages have not been widely accepted in mainstream software industry. For example, one of the most popular AOP languages, AspectJ, provides aspects, pointcut, and advice. These new language constructs are similar to classes and methods but different enough for developers to spend a substantial amount of time to learn. GluonJ is a language that provides AOP functionality by natural extension to OOP-based language constructs. Although GluonJ is similar to OOP languages, which are well known, developers still have to learn new language constructs. The Aspect-Aware Interface (AAI) is a new kind of interface for understanding crosscutting structures in a AspectJ program [50]. When a program is written in AspectJ, to understand the whole behavior of advised method will be difficult only looking at the implementation of the method. This characteristics of AOP is known as obliviousness. AAI addresses this problem to represent which programming interfaces are extended by aspects. However, AAI is a language construct that is based on the AOP languages. Therefore, Developers need to learn a AOP language to use AAI. Open Modules and XPIs (crosscutting programming interface) 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.

Classbox [13] are not AOP language constructs but are are modules that can provide a custom interface to selected clients. Although Classboxes provide better information hiding and modularity, AOP languages provide better expressiveness for describing conditional extensions (or custom interfaces in the terminology of Classboxes). Another approach to address the drawbacks of the obliviousness property is to introduce language constructs into AOP languages. There have been several constructs proposed on this approach: for example, open modules [6, 70] and XPIs (crosscut programming interfaces) [40]. Their idea is to let developers declare a module interface for pointcuts. They must explicitly specify selectable join points from external clients so that the fragile pointcut problem [56] can be avoided. The developers can take care of those selectable join points when they modify the implementation of the module. A disadvantage of this approach is that developers must anticipate join points that will be selected by aspects deployed in future. Anticipating all necessary join points in advance is difficult. Otherwise, developers must manually update module interface whenever new join points must be selectable.

There are lots of other modularization techniques in programming languages. MultiJava [30] provides a mechanism to define methods of a class from outside of the class. MultiJava shares the basic idea of the inter-type declaration of AspectJ. HyperJ [27, 81] can separate concerns separately and compose them freely according to a modular view that needs at that time. GluonJ [22] is a language that provides AOP functionality by natural extension to OOP-based language constructs. Although GluonJ is similar to OOP languages, which are well known, developers still have to learn new language constructs. Feature Oriented Programming [12, 10] is another way of modularizing concerns as *features*. Context Oriented Programming [78] is another language paradigm to treat program concerns as *contexts*.

Thus, the desire to modularise a new type of concern or realise a new type of abstraction often prompts the development of a new programming language (or extensions). Each such language promises to improve modularity, maintainability, and reusability in a particular way. Yet, with each new language (extension) come a variety of risks, including: the cost of learning the new constructs; risks of change in a company's business processes (e.g., to use AOP, a new process for aspect development is needed); and costs of refactoring previously developed systems.

It should also be noted, that the initially selected programming abstractions will have a significant effect on the modularity and subsequently the maintenance, particularly for concerns which are not well suited to the selected programming abstractions. For example, in OO class-based modularisation does not allow localised representations and treatments of crosscutting concerns. This in turn increases their maintenance costs.

5.2.2 Natural Languages for Modularity

There are a few pieces of work that use natural language text as basis for modular representation of concerns. One such work is the CommentWeaver. It reduces redundant repetition of text in an application API by physically modularising the repeated text. The text modules are then composed into full API specifications. Compositions use name or String-based pattern matching and reference the concrete programming structures that own APIs where the modularised text is to be composed. This work took an initial step of text modularisation in code, but used the physical code structure for composition.

Other related work is that on RDL [24] with its supporting MRAT tool [84]. This work resides in the domain of natural-language-based textual requirements only. Here natural language-based queries are used to explore requirements text, as well as to define composition for physically modularised crosscutting requirements. Unlike our proposal, this work does not use text as a proxy for the code semantics and structure.

Pegasus [52] is a new programming language that uses the help of the natural language. Pegasus fills in the gaps between the intention developers have at the first step and actual program structures they have to write then. Therefore, it enables to write developers' intention with the natural language instead of writing concrete codes. Its compiler process a description with the natural language into a concrete program. For example, a description "Write ten times: "pegasus"" will be processed into a concrete java program to print out the string "pegasus" for ten times. It has the similar motivation with the literate programming, but provides more concrete insight about the conjunction in documentation and programming. Authors mention about the applicability to OOP. For example, to realize the inheritance relation, the sentence "A student is a person." will be a clue to define it. Currently, Pegasus supports only number, character string, array etc in German and English. Lopes *et.al* also gave a discussion about documentation for programming [61].

5.2.3 IDEs for Modularity

Another approach to generated alternative modularity views is via IDEs. IDE visualisation tools are useful for understanding code concerns. For instance, to help understand AspectJ programmes, a developer may rely on AJDT as mentioned in chapter 3. Active Models [31] is another approach to represent a crosscutting structure better than AJDT. ActiveAspect, which is their tool based on the active models, presents a node-and-link diagram representing an interesting slice of the crosscutting structure of an AspectJ aspect. ActiveAspect's approach is to visualize join points selected by aspects.

Visual separation of concerns (VSC) [25] IDE tool provides modular views on crosscutting concerns to Java developers. Similarly, Code Bubbles [18] can be used to address the problem of source code navigation which can account to 35% of developers' time. Code Bubbles are used by searching across the source code and grouping the manually identified related code into a localised bubble. This method requires a developer to have good knowledge of the base code. Mylyn [47] attempts to eliminate this problem by creating modularisation based on performed tasks. Mylyn monitors the activities frequently performed by developers, and extracts the structural relationships of program artefacts. Then it creates localised views of code artefacts centred upon the developer's activities. Thus, most current IDE tool current tools either work by exploring the code directly, and/or are implementation language-dependent. More specific to AOP, Fluid AOP [43] allows the developer to switch to alternative crosscutting views to enable specific editing or reasoning tasks. The code could appear to have different crosscutting modularities simultaneously, as opposed to just having modules that crosscut each other. All of these IDE-based tools are implementation language-dependant, as well as developed for exploration of specific types modular views - crosscutting concerns.

To sum up, there are different ways of achieving modularity, most still using programming language constructs, or relying on such constructs for rendering alternative modularity views. A few approaches, particularly in requirements engineering, use NL, to achieve modularity based upon the semantics and syntax of that text. Our proposal, involves combining these two approaches: code structures (through IDE) and NL.

5.3 Towards Technology beyond AOP

Once comment is separated into each programming module such as methods and classes, a way of composing these separated comments is necessary. Especially, when precise composition results for generated documentation are needed, a set of composition mechanisms will be crucial. That mechanism should be written in the natural language and comprehensive against lots of kinds of programming languages. In addition, software is often updated through years and modified and changed. A requirement will be added, removed, or modified in some cases. In another case, a refactoring of the software might be just done. Writing documentation should be adopted these demands.

For this purpose, we use semantic-based comment weaving in the generation of documentation, which is robust against refactorings of software. To demonstrate the structure-independent nature of Universal AOP, we use Javassist [21] - a class library for bytecode transformation. As per one of the main functions of this library, several methods should be prepared for manipulating class bytecode. Some of these methods will convert a class definition to a class file, others will write out the class file into a local disk. One common concern of these methods is the fact that once each method manipulates a class file, further modification of that class file is not allowed. Since this common concern crosscuts all the methods related to class manipulation, it should be modularised in one location in the Javassist documentation. However, when HTML files are generated for Javassist API, this crosscutting concern should be documented in all the related method descriptions. Thus, the challenge is compose the localised crosscutting concern specification into the API documentation of several relevant methods.

We define this composition by using the grammar and semantics of the natural language documentation. The pointcut is defined with respect to the grammatical subject, verb, and object of the documentation sentences. For example, the pointcut to capture the doc comments of writeFile methods can be written as shown below. Two writeFile methods share a basic behaviour that is described as “Writes a class file” in their doc comments. Therefore, our pointcut specifies that “write” undertakes the role of verb (also termed relationship) and “class file” or “file” takes the role of grammatical object. Note that verb changes such as “writes”, “wrote”, and “writing” can be captured with our pointcuts. The important thing is that their behaviours of writing a file will not be changed through refactorings. Thus, once this pointcut are defined, there is no need to refactor along with implementation

Advice tags	Position of inserted text
@before	At the front of the sentences
@metBy	At the beginning of the selected sentence that includes matched text
@meet	At the end of the selected sentence that includes matched text
@after	At the end of the sentences

Table 5.1. Advice tags in Universal AOP

refactorings.

```
/**
 * @meet: relationship="write" and object="file" {
 *   Once this method is called, further modification of
 *   that class is not possible any more.
 * }
 */
void toBytecode() { ... }
```

All the keywords used for the pointcut definition in Universal AOP are shown in Table 5.2. While the subject, relationship (verb), and object refer to normal grammatical roles in a sentence, the including keyword is used to provide qualifying terms often used in a word phrase (e.g., "public method" is a phrase where method is qualified by public adjective). The categoryOf keyword allows reference to broad verb categories [33, 24], such as Move category, which includes such verbs, as run, throw, and post.

The @meet tag is used to insert the bracket text just after the selected text. In this example, the text starting "Once this method is called" is inserted after the text described as "writes as a class file". Currently provided tags are shown in Table 5.3. Universal AOP provides three other tags for inserting various points in sentences. The @metBy tag is used to insert the bracketed text just before the selected text. On the other hand, the before tag inserts text in the beginning of whole sentences in doc comments. Thus, there will be no text before the inserted text after comment weaving. To the contrary, the @after tag inserts text at the end of whole sentences.
As shown through an example above, the pointcut in our Universal AOP is less fragile against refactorings at the implementation phase. Here, we call a pointcut is fragile when the pointcut selects unintended doc comments or does not select intended doc comments. In addition, Universal AOP provides the other functions to make pointcuts less fragile.

Synonyms

One of such functions is to capture synonyms of a specified relationship. Thus, users of Universal AOP can specify widely the target word. For example, the former example of toBytecode can be redefined by using relationship="*compose*" instead of *"write"*. In order to allow synonyms, we use the WordNet[5], which is a large lexical database for English words.

Verb categories

When users of Universal AOP want to capture the broader meaning of verbs, they can also use verb categories, which are originally made by Ruzanna et al.[24]. This is a classification made from a set of verb classes and subclasses that cover all English verbs. For example, a verb group *mental action* includes *decide*, *discover*, *think* etc. Therefore, this is useful to capture broader meaning of verbs than synonyms do.

The other features

Universal AOP also enables user-defined dictionary to refer to the special meaning of words. For example, the noun *member* often means the methods, constructor, field etc. in the context of programming languages. In case that specifying each words are burdensome, developers can define the domain specific noun relationship in a dictionary file as follows.

```
member:=method,field,constructor,...
```

pointcut	examples
subject	subject="class" matches *"This class provides.."*
relationship	relationship="write" matches *"Writes a class file"*
	negative relationship="prune" matches *"This method does not prune ..."*
object	object="file" matches *"Writes a class file..."*
including	relationship="create" and object="method" and including="public" matches *"Creates a public method..."*
categoryOf (= categoryOf relationship)	categoryOf="sit down" matches *"... lie down ..."*

Table 5.2. The pointcuts available in Universal AOP

As another function, generic terms are available in the bracketed text of Universal AOP tags. They are transformed into a proper name when HTML files are generated. For example, if the term *thisMethodName* is included in the doc comment of the toBytecode method, the term is transformed into the concrete methodname toBytecode in the generated HTML files of selected documentation.

5.3.1 Structure and Language Independence

To illustrate the program structure and language independence of modular views, let us consider two examples, shown in Figure 5.2 and 5.3, where file contents are read. The purpose of this example is to demonstrate how to extract a code slice related to a particular concern by using NL comments. Let us call this slice (or alternative modular view) *LineReading*. We start building this concern by issuing an NL query "read line". Let us follow the process of building *LineReading* view.

For the example in Figure 5.2, the query "read line" will initially result in a NL comment in line 7 (Figure 5.2) identified as relevant. The view generator will then identify lines of code connected to this comment. Presently[2] we consider code to be related if it is directly connected to the comment (i.e., located immediately after the comment) or if it manipulates variables used in the directly connected code. For instance, in Figure 5.2 the identified comment is directly connected to the System.out.println(dis.readLine()) statement. For this statement the "dis" variable declaration (line 2, Figure 5.2) and initialisation (line 4, Figure 5.2) are relevant. In turn, declaration of "file" (line 1 Figure 5.2) is relevant to "dis" (line 4, Figure 5.2), and so is relevant to the initially commented line. The relevant code is then presented in the IDE as the *LineReading* view based on "read line" query. Statements unrelated to this view will not be included (e.g., the statements related to exception handling). All that the developer uses to construct the view, is a NL language query for "read line".

5.3 shows a Haskell functional program to again read the contents of a file. Yet, to construct a modular view for the *LineReading* concern, we can use the same NL "read line" query. It will identify the comment on line 4 in Figure 5.3 and the concern view will be constructed using the same rules of relevance as above. In this case, the comment belongs to the "System.IO¿hGetLine h". Thus, declaration of h (line 2 Figure 5.3) is deemed relevant.

[2]These are initial relevance rules and they are subject to evaluation and further re-

```
File file= new File(filename);
DateInputStream dis=null;
try {
   dis= new DataInputStream(new BufferedInputStream(new FileInputStream(file)));
   //available() returns 0 if the file has no more lines.
   while(dis.available()!=0){
      //this statement reads the line from the file and prints it to the console.
      System.out.println(dis.readLine());
   }
   //dispose all the resources after using them.
   dis.close();
}catch(Exception e){
e.printStrackTrace();}
```

Figure 5.2. OO LineReading concern example.

```
--Returns a handle onto the file "A.hs"
System.IO> h <- openFile "A.hs" ReadMode
{handle: A.hs}
--Read a line from this handle
System.IO> hGetLine h
"main=do"
--Close a handle, and flush the buffer
hClose :: Handle-> IO()
```

Figure 5.3. Functional LineReading concern example.

5.4 Impact of Modular Views

Since modularizing documentation is free from concrete programming structure, developers can use their favorite programming languages as long as they manage documentation with Universal AOP. They can achieve the same effect as AOP languages provide. Universal AOP has the three benefits that AOP languages provide.

- Modular view
  Generated HTML documentation contains hyper links to a concern description. This will help developers to understand crosscutting structures. For example, the text starting with "Once this method is" is woven into the documentation of the writeFile method with a hyper link. This hyper link jumps into the documentation of

search.

the toBytecode(DataOutputstream) method to inform that the concern described by "Once this method is" is implemented in toBytecode(DataOutputstream).

- Homogeneity
  This property is achievable by the Universal AOP tags. The description about a concern can be written in one documentation in source codes and it will be woven into the appropriate documentation when the HTML documentation is generated. Even though each implementation about a concern is crosscutting in OOP, documentation about the concern is still well modularized and developers can manage such concerns.

- Obliviousness
  This property is achievable with the different type of obliviousness on documentation although extending program behavior is currently out of our scope. A description about a concern is implicitly woven into the generated HTML documentation. For example, there is no description in the documentation of the writeFile method to indicate that further modification about a class definition is not possible.

5.5 Case study

In order to evaluate the weaving mechanism, we executed a case study by using Javassist library with version 2.6 and version 3.1. The metrics is shown in table 5.3. To modularize documents within source codes, developers have to put tags initially into these documents even if they choose to use either CommentWeaver or Universal AOP. Figure 5.4 shows the number of tagging that was needed initially in CommentWeaver. The @quote tag was needed to refer to doc comments of other methods. Since the @quote tag is used for a reference to a doc comment, tagging is needed more than the @export and @weave tags. As a result, 37 tags of @quote were needed in version 2.6 in CommentWeaver. The @export tag is needed to decide which text should be referred from the @quote tag. In other words, the number of tagging of the @export tag is the number of modularized document by @export, and the number of tagging of the @quote tag is the number of woven document. In this case, 22 @export tags were needed. Another way of weaving document is to use the @weave tag, which pushes document into several API documentation

Attribute	Metric	Description
Description of tagging	Number of initial tagging	- Measures how many tag elements are introduced
	Number of tagging added/changed/removed	- Measures How many compositions are altered during a maintenance change
Description of pointcuts	Precision	- The ratio between the number of relevant doc comments selected by a pointcut and the total number of doc comments selected by a pointcut

Table 5.3. Metrics suite summary

of methods that are selected by the argument of the @weave tag. The number of @weave used in version 2.6 was 21.

Total number of tagging in Universal AOP was 40, and it was less than that of CommentWeaver. As shown in Figure 5.5, the @after tag was mostly used because there were many cases that should weave the @throws tags into the API documentation and the @after tags specified the verbs and objects that are written in the first sentence in documents. If the @meet tag were used in such a case, the @throws tag would be woven into just after the selected sentence and generated API documentation would be murky. All the tags in Universal AOP push documents into the API documentation, and they correspond to the @weave tag in CommentWeaver. Universal AOP does not have a mechanism to refer to other documents as the @quote tag does. Instead, more precise weaving position can be specified in Universal AOP by using four kinds of tags as mentioned in Table 5.3. Note that since the tags in Universal AOP can decide which documents should be woven by bracketing these documents, they also have roles of the @export tag. Since, in CommentWeaver, the total number of tagging of the @quote and @weave tags was 58, tagging in case of Universal AOP resulted in 32% reduction.

Figure 5.6 represents the numbers of tagging that had to be added, removed, and modified by refactorings through version 2.6 to version 3.1. The left chart in Figure 5.6 shows the tagging that had to be newly added in version 3.1. The @quote tag was required in 6 documents and @export tag was required in 5 documents, which means that modularized documents were at least 5. Note that a reference by the @quote tag does not necessarily need the @export tag when a whole doc comment is referred. On the other hand, the @weave tag was necessary in one doc comment. The middle chart in Figure 5.6 shows the number of tagging that had to be removed in version

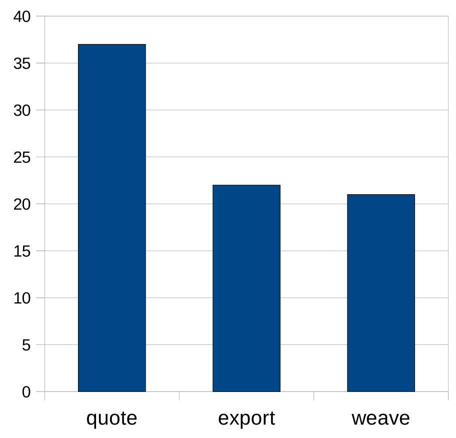


Figure 5.4. The number of tags that need initially for CommentWeaver

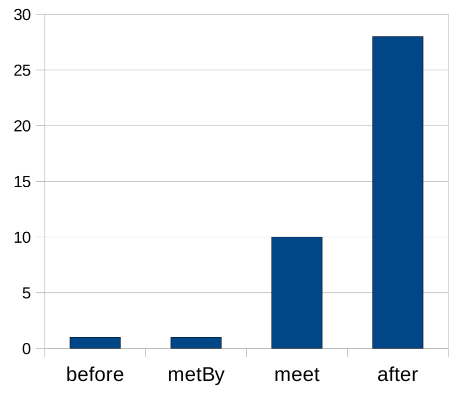


Figure 5.5. The number of tags that need initially for Universal AOP

3.1. This means that 9 methods were removed in version 3.1. Although the @weave tag was not removed, several target doc comments (and its methods) that would be woven into from the @weave tag were removed. The right chart in Figure 5.6 represents the number of tagging that had to be modified in their declarations. Modification in this case means that the part of the tag declaration need to be modified such as the parameters of @quote and @weave tag. Thus, there is no overlapped counting among modified tagging and added/removed tagging. The @export tag was not modified because once modularized doc comments did not need to be changed.

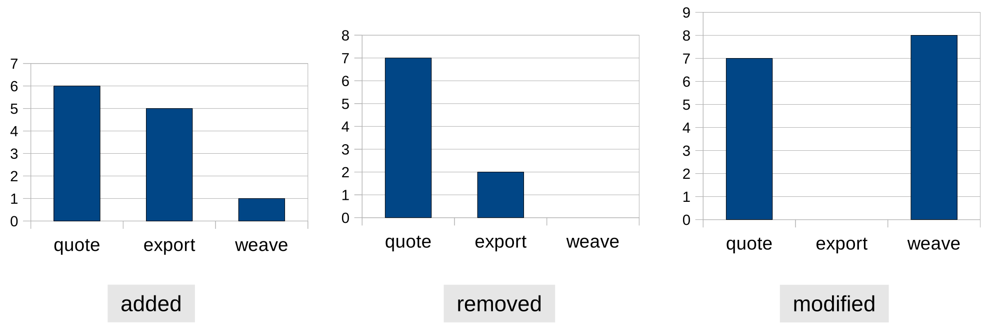


Figure 5.6. The number of tags added/removed/modified through ver.2.6 to ver.3.1 in CommentWeaver

Figure 5.7 represents the number of tagging that was added, removed, and modified in Universal AOP. Compared to the case of CommentWeaver in Figure 5.6, the total tagging was less needed (50% reduction). The left chart in Figure 5.7 shows the number of tagging added in version 3.1 in Universal AOP. In this case, the total tagging is the same number between CommentWeaver and Universal AOP (0% reduction). The middle chart in Figure 5.7 shows the number of tagging removed, and was less than CommentWeaver (45% reduction). The left chart in Figure 5.7 shows the number of tagging modified, and the result was 7 in total (53 % reduction). As mentioned above, modification does not have overlap with addition and remove in terms of counting the number of tagging. Since four kinds of tags are available in Universal AOP, a modification can be executed from a tag in these four kinds of tags to another kind of tag. However, there was no such a case in Javassist.

Table 5.4 shows the precision of comment weaving. In version 2.6, Universal AOP wrongly selected 5 documents (4 %). In version 3.1, it selected 4 documents unnecessarily (4 %). Although the precision was low in this case

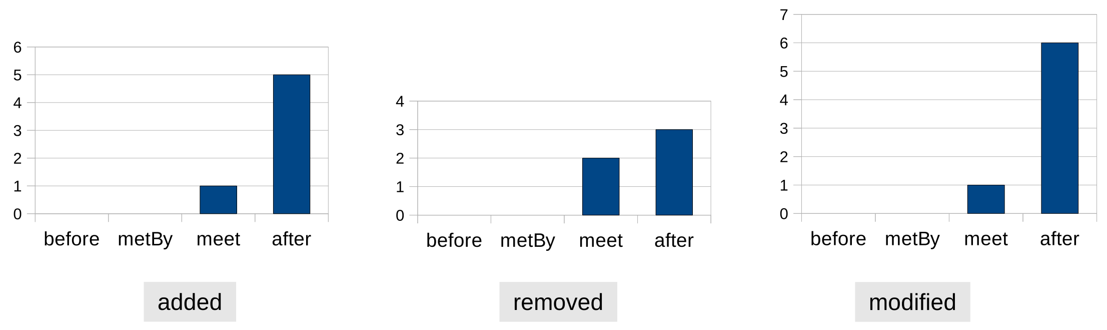


Figure 5.7. The number of tags added/removed/modified through ver.2.6 to ver.3.1 in Universal AOP

study, several reasons could be thought. Firstly, the documentation of Javassist is well written, thus, specifying crosscutting concerns by semantic-based pointcut is easier. Second reason would be the size of written documentation of Javassist. The lines of codes of documentation of Javassist is approximately 10,000. If the size were larger than this, the accuracy of woven result would be lower.

An example of defined pointcut in this case study is as follows. The documentation about the RuntimeException is modularized, and this will be woven into the specification that has *set* an *exception*.

```
@after: relationship="set" and object="exception" {
    @throws RuntimeException if the declaring class is frozen
}
before(CtBehavior cb):
     execution(void CtBehavior.setExceptionTypes(CtClass[]) {
   checkModify(...);
}
```

As a result, an unintended sentence as shown below was selected by this semantic-based pointcut. The first sentence was unintended but selected one. Second sentence below was one of the intended specifications. The semantics of these two sentences are different. However, current Universal AOP cannot distinguish these two sentences with the definition of semantic-based pointcut.

Sets the names of exceptions ...

Precision	ver.2.6	ver.3.1
Number of precisely selected document	133	103
Number of selected document	138	107
Ratio	0.96	0.96

Table 5.4. The ratio of the precision in weaving documents in Universal AOP

Sets exceptions that this method ...

Currently, developers have to add including not to select the unintended sentences by using the noun names as shown below.

```
@after: relationship="set" and object="exception" and !including="name"
```

5.6 Future research directions

In order to fully achieve our vision, various obstacles and additional work still needs to be performed. One of the most significant obstacles surrounds the effects the underlying modularity will have on the generated modularity views. The underlying modularity technique will influence how the comments are written. For example, in Java developers will typically write comments on a per-method basis, with the comments summarising and describing the behaviour of the associated method. Even though it will be possible to generate an alternative modularity view based on these comments, it may not be possible to more accurately ascertain which part of the method relates to which part of the comment. As a result, it will be necessary to include all of the code within that method within the newly generated modularity view. This is obviously not an ideal solution as potentially more code will be included in the view than necessary. Instead, it may be necessary to analyze comments embedded within method bodies to more accurately determine which parts of method should be included in new modularity view. This also leads to research having to be performed on how comments should be written for the modularity views to be accurately generated. In an ideal world no specific strategies should have to be applied when writing comments for them to be effective. However, this may not be the case. Studies will be performed that involve pre-existing systems that have multiple modularity views (for example, HealthWatcher [39] has an OO and AO implementation). Our approach can be applied to each implementation and assessment can be performed to determine whether the alternate modularity view can be accurately generated. Documentation-based modular views are a way of abstracting the concerns contained in the code from the code structure. We have demonstrated that NL documentations and comments can be used for extracting these concerns without reference to their implementation code. There are a number of open issues, such as: what is the best way of writing the comments/documentation in order to maximize the usability of these comments for modular concern views? A related work [23] shows that short sentences with clear stated subject, verbs, objects are amenable for automated annotation with good precision and recall. However, further evaluation of these and guidelines for writing such comments are needed.

5.7 Summary

This chapter presents our new documentation tool named *Universal AOP*, which is available with semantic-based comment weaving, that is, the natural language such as subjects, verbs, or objects is used to weave doc comments. Our documentation tool also provides the modular view of crosscutting concerns that can exist in OOP programs. Therefore, AOP languages are not necessary any more as long as Universal AOP are used instead. Developers just need to write precise documentation, which is always necessary to let user programmers (or developers themselves) use the software in practice. Through our case study with Javassist, Universal AOP contributed to less tagging into doc comments. The number of tagging resulted in half of tagging in CommentWeaver.

Chapter 6

Conclusion

This thesis has discussed tools for modularizing documentation at the implementation and design phase. When developers implement programs in AOP, AspectScope shows how classes are extended by aspects through documentation view. Since programming is the main pillar when implementing, AspectScope automatically generates woven documents. When software is released as APIs, CommentWeaver is available to generate the precise API documentation by using tags that specify concrete programming structures. When a refactoring is necessary for maintenance, Universal AOP provides semantic-based document weaving that does not require any concrete programming structures.

Contributions

The contributions by this thesis are summarized as follows:

- This thesis presents that documentation contains non-negligible crosscutting concerns, and existing tools and languages such as Javadoc and the literate programming do not enable modularly describing the documentation. It shows that this problem often occurs in object oriented programming, and aspect oriented programming makes this problem more complicated.

- Then, this thesis proposes aspect-oriented documentation tools for modular description of documents. We prepared three scenarios: implementing programs, publishing the API documentation, and doing maintenance for once released programs. Each tool are developed under the criteria: the precision of woven documents and the amount of tagging.

- This thesis also discusses another benefit of modularizing documentation. Our semantic-based document weaving provides modular views of programming concerns as well as AOP languages do. This means that our tool is alternatively available for separating crosscutting concerns in programs as long as documentation are well written and managed.

- This thesis illustrates the applicability of our tools by using several kinds of software. They are widely used libraries and frameworks such as Java 6 library, Eclipse framework, and Javassist library.

Future Directions

Possible future directions of this thesis are as follows:

Constructing sophisticated modular views for semantic-based weaving Currently, the woven results of documents by Universal AOP can be viewed through the API documentation. The hyper link connects each concern on the HTML format. Although this modular view make developers recognize crosscutting concerns, more sophisticated views should be constructed by using Eclipse IDE etc.

Conducting large-scale case studies for semantic-based weaving For the evaluation of Universal AOP, we used the Javassist library. Although this case study brought the result that semantic-based weaving is useful for modularizing documentation, larger-scale case studies such as Java 6 library and Eclipse framework should be conducted.

Eliminating an ad-hoc approach of the part of semantic-based weaving Universal AOP uses several tags to refer to the specification of programs. One of these tags such as the including tag is currently necessary to narrow down selected documents because there are a lot of similar sentences in documents. However, if confirmation of the precise of woven documents is somehow possible, this ad-hoc approach such as using the including tag should be avoided.

Bibliography

[1] AspectJ Development Tools(AJDT). http://www.eclipse.org/ajdt.

[2] AspectJ project. http://www.eclipse.org.aspectj/.

[3] Eclipse Bug Reports. https://bugs.eclipse.org/bugs/.

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

[5] WordNet, 2006. http://wordnet.princeton.edu.

[6] 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, 2005.

[7] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 345–364, 2005.

[8] Kellens Andy, Mens Kim, Brichau Johan, and Gybels Kris. Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[9] Sven Apel and Don Batory. When to use features and aspects?: a case study. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 59–68, 2006.

[10] Sven Apel and Don Batory. When to Use Features and Aspects?: A Case Study. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 59–68, 2006.

[11] A. Avenarius and S. Oppermann. FWEB: A Literate Programming System for Fortran8x. *SIGPLAN Not.*, 25(1):52–58, 1990.

[12] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 177–189, 2005.

[14] 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, 2007.

[15] 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, 2006.

[16] B. W. Boehm. *Software Engineering Economics*, pages 641–686. Springer-Verlag New York, Inc., 2002.

[17] Michael D. Bond and Kathryn S. McKinley. Probabilistic Calling Context. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 97–112, 2007.

[18] Andrew Bragdon. Developing and Evaluating the Code Bubbles Metaphor. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 525–526, 2010.

[19] Walter Bright. The D Programming Language. *Dr. Dobb's J.*, 27:36–40, February 2002.

[20] F. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20:10–19, April 1987.

[21] 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.

[22] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 539–554, 2010.

[23] Ruzanna Chitchyan, Phil Greenwood, Americo Sampaio, Awais Rashid, Alessandro Garcia, and Lyrene Fernandes da Silva. Semantic vs. Syntactic Compositions in Aspect-Oriented Requirements Engineering: An Empirical Study. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 149–160, 2009.

[24] Ruzanna Chitchyan, Awais Rashid, Paul Rayson, and Robert Waters. Semantics-Based Composition for Aspect-Oriented Requirements Engineering. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 36–48, 2007.

[25] Mark C. Chu-Carroll, James Wright, and Annie T. T. Ying. Visual Separation of Concerns through Multidimensional Program Storage. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 188–197, 2003.

[26] Mariano Cilia, Michael Haupt, Mira Mezini, Alejandro Buchmann, and Ro Buchmann. The Convergence of AOP and Active Databases: Towards Reactive Middleware. In *In Proc. GPCE 2003*, pages 169–188. Springer, 2003.

[27] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 325–339, 1999.

[28] Curtis Clifton and Gary T. Leanvens. Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning. Technical report, Iowa State University, 2002.

[29] Curtis Clifton and Gary T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *FOAL 2002*, 2002.

[30] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 130–145, 2000.

[31] Wesley Coelho and Gail C. Murphy. Presenting crosscutting structure with active models. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 158–168, 2006.

[32] 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.

[33] R. M. W. Dixon. *A Semantic Approach to English Grammar*. Oxford Textbooks in Linguistics. Oxford University Press, New York, 2005.

[34] Rémi Douence, Pascal Fradet, and Mario Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 173–188, London, UK, 2002. Springer-Verlag.

[35] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 141–150, 2004.

[36] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An Expressive Aspect Language for System Applications with Arachne. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 27–38, 2005.

[37] Rémi Douence and Mario Südholt. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical report, 2002.

[38] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[39] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP '07 : Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 176–200. Springer-Verlag, 2007.

[40] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. volume 23, pages 51–60, Los Alamitos, CA, USA, January 2006. IEEE Computer Society Press.

[41] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, 2003.

[42] Crhis Dutchyn Hidehiko Masuhara, Gregor Kiczales. Compilation semantics of aspect-oriented progrmas. In *Proceedings of Foundation of Aspect-Oriented Languages Workshop*, AOSD'02, pages 17–26, 2002.

[43] Terry Hon and Gregor Kiczales. Fluid AOP Join Point Models. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 712–713, 2006.

[44] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 318–326, 1997.

[45] 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.

[46] Michael Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '98/FSE-6, pages 131–142, 1998.

[47] Mik Kersten and Gail C. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 159–168, 2005.

[48] 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.

[49] 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, 1992.

[50] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, 2005.

[51] Ostermann Klaus, Mezini Mira, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 214–240, 2005.

[52] Roman Knöll and Mira Mezini. Pegasus: First Steps Toward a Naturalistic Programming Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 542–559, 2006.

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

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

[55] 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.

[56] Christian Koppen and Maximilian Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *European Interactive Workshop on Aspects in Software(EIWAS'04).*

[57] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[58] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

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

[60] 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.

[61] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond AOP: Toward Naturalistic Programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 198–207, 2003.

[62] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs, 2002.

[63] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[64] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

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

[66] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 52–67, 2002.

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

[68] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 99–115, 2004.

[69] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 21–36, 2006.

[70] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to aspectj. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 39–50, 2006.

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

[72] 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.

[73] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

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

[75] Hridesh Rajan and Kevin Sullivan. Aspect Language Features for Concern Coverage Profiling. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 181–191, 2005.

[76] Awais Rashid, Ana Moreira, and João Araújo. Modularisation and Composition of Aspectual Requirements. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 11–20, 2003.

[77] Steven P. Reiss. Simplifying Data Integration: The Design of the Desert Software Development Environment. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 398–407, Washington, DC, USA, 1996. IEEE Computer Society.

[78] Hischfeld Robert, Costanza Pascal, and Nierstrsz Oscar. In *Journal of Object Technology.*

[79] Kouhei Sakurai and Hidehiko Masuhara. Test-based pointcuts for robust and fine-grained join point specification. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 96–107, 2008.

[80] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11:215–255, April 2002.

[81] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, 1999.

[82] Cottenier Thomas, van den Berg Aswin, and Elrad Tzilla. Joinpoint Inference from Behavioral Specification to Implementation. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 476–500, 2007.

[83] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, 1987.

[84] R. W. Waters. *MRAT: A Multidimensional Requirements Analysis Tool.* MSc dissertation, Lancaster University, 2006.

[85] Elcin Recebli Wolfson. Pure aspects. Master's thesis, Oxford University, 2005.