An Alternative Use of AOP to Facilitate Program Understanding

Jeff Ditullio Chandra Krintz Lingli Zhang

Computer Science Department
University of California, Santa Barbara
{jditu,ckrintz,lingli_z}@cs.ucsb.edu

Abstract

With this paper, we present a new way of employing Aspect-oriented techniques to facilitate source code understanding by programmers. AOP is commonly used to add functionality to programs at well-defined points in the code and to separate out cross-cutting concerns. We use AOP to identify opportunities to *hide* unrelated code regions from view for software that embeds cross-cutting concerns.

To enable this, we define new extensions to AspectJ in the form of pointcuts for conditional branch instructions that we refine with string names of variables and fields used in the conditional expression. We define a new declare action that makes use of these pointcuts to mark bytecode instructions. We extend the Eclipse IDE with a plugin that consumes these bytecode markers, identifies their source equivalent, and hides the corresponding code blocks. Our system automatically generates the necessary aspects that enable AOP-guided folding, using three different mechanisms: a database of commonly used names, profile data that identifies frequently and infrequently executed code bodies, and domain-specific information. We show using examples and empirical data that our system reduces the number of irrelevant source lines that a user must view when investigating Java source packages via a visualization system.

1. Introduction

The popularity of and support for open source technologies have lead to the emergence of an increasingly large number of available software systems. Such systems, include virtual execution systems (e.g., runtimes, operating systems), integrated development environments, web servers, games, email clients, and others. However, due to the complexity of these systems, it is difficult for those that want to extend them to understand the intricacies of the functionality without a significant learning curve. This is especially true, for students in a University setting in which open source software tools are studied and extended as part of course-work [15, 18, 19]. Such systems offer a tremendous opportunity for a students' education, experience, and exposure to real-world infrastructures. However, to enable this, we require tools that reduce the learning curve for understanding and modifying these software systems. Such tools increasingly play a key role in the implementation of modern programming languages.

One such technique for improving programmer understanding of software, that has gained recent popularity, is *aspect-oriented programming (AOP)* [14]. With AOP, programmers disentangle cross-cutting concerns by specifying them as a separate abstraction called an *aspect*. Automatic aspect-aware tools then weave the aspect into a program at the appropriate points in the code. For example, programmers commonly insert code for debugging, collecting timing or state data, or to assert certain conditions, into methods. Using AOP, the debugging, logging, and assertion code is implemented within individual aspects. The programmer specifies the points in the code at which each aspect should be inserted. When the programmer builds the program, a weaving tool performs the appropriate insertions. AOP enables programmers to write truly modular code in which the modules implement a single, self-contained functionality. This modularization promotes code understanding by programmers learning the software since the algorithms and functionality implemented by methods is unobscured.

One limitation of AOP however, is that it aids programmers in the development of *new software*; it does not help to improve *existing* software, unless that software is rewritten to use aspects. Unfortunately, most software contains unrelated, nonessential, and cross-cutting code within method bodies, that impede a new user's ability to understand what the software is doing. Figure 1 shows and example of this type of programming style from an snippet of code taken from a source file that is part of the Jikes Research Virtual Machine (JikesRVM) [2], an open-source JVM that is written in Java. The instruction in the dotted box is the only line of source code that implements the method's functionality; all other code blocks are cross-cutting concerns.

To address this limitation, we extend AOP for Java programs for use in a non-traditional way. In particular, we use AOP to identify points in the source code that are not pertinent to the underlying algorithm of a method – and remove them from view. To enable this, we extend AOP with new pointcuts that identify conditional branch bytecode instructions. Our implementation refines these pointcuts by matching arbitrary strings, such as those used for variables and methods, against those used in the conditional expression. In addition, we implement AOP actions that use of these pointcuts to place markers at (i.e., before, after, and around) each instruction identified.

We then use this AOP extension within the Eclipse Integrated Development Environment (IDE) framework to hide unrelated code from view. Our Eclipse extensions consume the markers from a bytecode aspect and uses them to *fold* the source code out of a user's view. Our folding technique creates a new perspective of the code *without* changing the semantics of the program. This semantic retention is key to our approach since it enables novice users of software to reap the benefits of AOP with little effort. Given the JikesRVM example in Figure 1, our system automatically folds all code except the line in the dotted box.

```
private static VM_CompiledMethod optCompile(VM_NormalMethod method,
  OPT_CompilationPlan plan) throws OPT_OptimizingCompilerException
{
   if (VM.VerifyAssertions) {
       VM._assert(compilationInProgress,
           "Failed to acquire compilationInProgress \"lock\"");
   }
   VM_Callbacks.notifyMethodCompile(method, VM_CompiledMethod.JNI);
   long start = 0;
   if (VM.MeasureCompilation || VM.BuildForAdaptiveSystem) {
      start = VM_Thread.getCurrentThread().accumulateCycles();
   }
   VM_CompiledMethod cm = OPT_Compiler.compile(plan);
   if (VM.MeasureCompilation || VM.BuildForAdaptiveSystem) {
       long end = VM_Thread.getCurrentThread().accumulateCycles();
       double compileTime = VM_Time.cyclesToMillis(end - start);
       cm.setCompilationTime(compileTime);
       record(OPT_COMPILER, method, cm);
   }
   return cm;
}
```

Figure 1. Code snippet showing how cross-cutting concerns obscure the underlying functionality of a method. This is code from a class in the compiler infrastructure of the open-source JikesRVM system.

We also automate the process of writing the necessary aspects for code folding. A user opens a project with source files, adds a default aspect (that our system automatically generates) to the project, and builds the project. During compilation (from source to bytecode), the system applies the aspects to produce the necessary markers which Eclipse uses to identify locations in the source code to fold.

We investigate three ways to employ this methodology to focus the code regions that a user visualizes. We consider commonly used names, profile data, and domain specific information. We implement our techniques within the open-source Eclipse IDE and AspectJ framework and show examples of its efficacy. In addition, we measure the impact of our approach in terms of lines of source visible and folded. Our experiments, show that our simple techniques are able to eliminate approximately 10% of irrelevant source code from view with little help from the user.

In summary, we make the following contributions with this paper:

- We extend AOP with new pointcuts and actions that mark instructions in the Java bytecode of a program.
- We employ our AOP implementation to extract code virtually (i.e., fold away visually) that is not pertinent to the overall algorithm a method implements.
- We enable automatic pointcut generation that is guided by novel techniques including code block selection via common programming idioms, program profile information, and domain-specific information.
- We implement our approach within the freely available Eclipse IDE framework and AspectJ.
- We evaluate the efficacy of our approach on the readability of code in terms of the number of source lines folded and the number of pertinent source lines that are made viewable.

In the sections that follow, we overview our approach and describe our prototypical implementation within Eclipse and AspectJ. We then evaluate the efficacy of folding on source visualization (Section 6). In the remaining sections, we present related work (Section 7 and our conclusions (Section 8).

2. Using AOP to Remove Cross-Cutting Concerns from View

The goal of our work is to improve ease with which large-scale software systems are understood by new developers. Many such systems implementing a wide range of technologies are increasingly available as open source, e.g., operating systems, virtual execution and software development environments, web services, etc. These systems have the potential for wide-spread use and extension in industry, research, as well as in the classroom. However, such systems are difficult to understand and extend even when they are written in object-oriented programming languages such as Java, since methods commonly contain significant amounts of cross cutting concerns or unrelated code blocks.

Programmers insert such blocks to assert that specific conditions hold, to measure the performance of a code region, to print out debug statements, or to log the state of the process at that point in the execution. In our experience, a common implementation of these cross-cutting measures is to insert these code blocks using if-statements that check for a certain flag before conditionally executing the region. For example, the open-source JikesRVM system implements 110 methods (of size 30 source lines or more [1]) for its implementation of the execution environment (methods from VM_* classes). Within these methods, 10% of the lines are within code blocks of if statements for which the conditional includes the string `debug`, `verbose`, `measure`, or `log`. These strings are names of local variables and fields that the programmer used to guard the execution of cross-cutting concerns. It is this type of programming style that is the focus of our work.

Aspect-oriented programming (AOP) offers a solution to this problem. Programmers specify cross-cutting concerns separately as aspects and the compilation system *weaves* in these code blocks as the appropriate points in the compiled code. Users specify potential points in the code using *pointcuts* which the compilation identifies as *join points* when conditions specified in the pointcut hold.

AOP offers a framework that enables programmers to improve the modularity of software as they write it. However, much aspect-unaware software exists; to achieve the benefits of AOP, these programs must be retrofitted with aspects. Much recent research has focused on automating this retrofitting process and is referred to in the literature as aspect mining [9, 10, 24, 1, 4] and AOP refactoring [6, 11, 17]. Aspect mining is the process of automatically (or with the help of programmers) identifying cross cutting concerns in software. AOP refactoring is the process of extracting the concerns identified via mining and automatically creating aspects from them. Extant approaches to aspect mining and refactoring are resource intensive, require source code for the entire system, and require expert programmer participation to ensure that the desired outcome (the resulting code partitioning) is achieved.

In our work, we present a system for *virtual* mining and refactoring that enables similar results without modifying the program, imposing overhead, or requiring programmer expertise. Moreover, we employ AOP itself to automate this process.

In our system, users specify AOP abstractions that enable an integrated development environment (IDE) to remove from view, the cross cutting concerns in code. That is, we employ AOP in a non-traditional way – to hide unrelated code blocks. Our AOP extensions are not *woven* into the compiled code but instead mark points in the the code that map to source code lines. Our IDE extensions then use these markers to automatically fold code blocks that are delimited with curly brackets and parentheses, syntax commonly used to specify conditional execution of cross-cutting code bodies. Our techniques are only modifications to the perspective

[1] We consider only those methods that span more lines than can be viewed in an Eclipse IDE source window. We selected 30 lines for this size.

that the user is focused on – they do not change the semantics of the program.

We implement our system within the Eclipse framework [8] which implements AspectJ via the ajc compiler [13] as plugins. We perform instruction marking online, during incremental (or batch) compilation of the program, as is currently done for other AspectJ aspects. We then modify the view of the source program on-the-fly. As such, our techniques or folding as well as for the identification of points in the code at which folding should occur are efficient. Moreover, our system does not require that the user be familiar with the code she is viewing, in order to reap the benefits of our AOP-guided source folding system.

We first overview the relevant parts of the AspectJ system that we extend and then describe the various components of our system. Our system includes AspectJ extensions for identifying specific conditional branch code blocks and for marking these instructions in the bytecode. The second component is an extension to the Eclipse IDE that exploits these markers to fold away from view the cross-cutting concerns that they demark. The final component of our system is a set of techniques that identify instructions to mark, and thus fold, using aspects that we generate automatically.

3. AspectJ

AspectJ is an extension to the Java programming language that enables users to define cross-cutting concerns separately and independently. The AspectJ system interleaves i.e., *weaves*, the code that implements the different concerns into the program automatically at the appropriate points. Users specify these points and the operations that must occur. AspectJ refers to as dynamic events in the program at which weaving can occur as *join points*.

The join points that AspectJ currently supports are:

- Method calls and executions,
- constructor calls and executions,
- class field access,
- execution of exception handlers, and
- class or object initialization.

Users use the *pointcut designator* abstraction (pointcuts for short) to select program join points. Pointcuts filter uninteresting events. Examples of pointcuts include *call*, *execution*, *get*, *set*, and *staticinitialization* pointcuts. AspectJ also provides pointcuts that restrict static (*within* for "within the execution logic of a particular type") and dynamic (e.g., *cflow* for "within in the control flow of") scope. Users can combine pointcuts using common logical operators, e.g., ||, &&, and !.

AspectJ implements the join point *kinds* above since they cover a wide range of implementations of cross-cutting concerns. In addition, these points are straightforward to identify and to add code in bytecode – the level at which weaving is performed by most systems [13, 3, 5]. However, this course-grain specification of join points limits finer-grain AOP, e.g., at the basic block, loop, or instruction level. Researchers have extended the AOP systems to enable point cuts at the basic block level to enable program testing in the Microsoft .Net framework [20] and to identify potentially parallel loops in AspectJ [12]. To enable our work, we implement a new pointcut in AspectJ for a particular bytecode instruction.

The user also specifies the actions that are taken (and when they are taken) when a selected join point executes. AspectJ supports advice that is inserted *before*, *after*, and *around* (both before and after). The user implements code that is executed at the join points depending on the *kind* of advice. The weaving process interleaves this code with that of the original program guided by the advice for a selected join point.

AspectJ supports a second type of action called `declare` statements. Declare statements enable static-crosscutting, e.g., introduction of supertypes (`declare parent`), or alteration of a method's exception specification (`declare soft`). Other declare statements are a compile-time mechanism that programmers can employ to force the ajc compiler to communicate with the user or IDE. For example, AspectJ supports `declare error` and `declare warning` in which the compiler sends a message to the execution environment (or standard error) about the source lines at which a pointcut expression will match. AspectJ defines these statements to enable users to enforce design constraints and to ensure that modularization is maintained as the program is modified. We extend this mechanism to enable interaction with the Eclipse IDE for our implementation of folding.

4. Extending AspectJ to Enable Automatic Source Folding

We extend AspectJ with a new pointcut to which we refer to as an `ifCheck` pointcut. In addition, we add the necessary supporting advice for this pointcut as well as a new type of declare statement to facilitate fine-grain communication with the Eclipse IDE to enable folding of source lines. We describe these extensions in the following sections. We then detail the various ways in which we specify (and automate specification of) the code that should be folded.

4.1 The `ifCheck` Pointcut

The `ifCheck` pointcut matches all conditional branch bytecode instructions that use values of fields or local variables as part of their boolean expression. We use the strings to match against field and variable names of the conditional branch expressions. The format of this pointcut pattern is as follows:

```
ifcheck(VarType [DeclType]VariableName);
```

VarType is the type of the field or local variable used in an expression of a conditional branch. VariableName is the name of the variable and DeclType is the class in which the field is defined (and is ignored if the variable is a local). Users can specify the wildcard (*) for the components of the `ifCheck`. For example, given the pointcut

```
ifcheck(* *DEBUG*);
```

our system matches a conditional branch bytecode with a condition that employs any field or local variable of any type with a name that contains the string "DEBUG".

The Eclipse AspectJ system parses and compiles both .java and .aj files to standard java class files incrementally as the user modifies the programs, saves the programs, and loads the class files of programs (the user can also choose to turn off incremental compilation in which case, the system compiles the files in a project when the user selects batch compilation). The system stores aspects as class files and advice (including declare statements) as methods. The system parses pointcuts within advice and declare statements and records each as method attributes [16] in the class file.

Each pointcut instance implements a *signature* that consists of the pattern that the programmer specifies. In AspectJ, the signature template includes the modifiers, the return type, the declaring type, the name, the parameter type list, and the throws pattern. The parsing process fills in the relative parts of the signature for each pointcut; a special type called "ANY" is specified for irrelevant fields or when the wildcard is specified in the pointcut.

The AspectJ plugin in Eclipse implements a *PatternParser* that the system uses to parse pointcuts. We extend this utility to recognize the `ifCheck` pointcut that we define above. Our extensions

fill in the signature template of the pointcut. We use the return type slot for the actual type of the variable. If the declaring type is not specified or is specified by a wildcard "*", our parser extensions fill in the declaring type slot with the "ANY" type. This means that the pointcut can be matched to both class fields and local variables. We also create a new pointcut object from this signature and set its kind flag to `ifCheck`. We serialize this object into the method attribute of the advice to which it is linked in the aspect.

Once the system compiles the source files to classes, it feeds them to the weaver. The weaver creates a *munger* for each advice (or declare statement). The weaver extracts the pointcut specifications from the advice attribute and stores it in a corresponding munger. When this process completes, the weaver has a list of mungers which it uses to compare to each class file (i.e., method bodies) for potential join points. For example, a *GETFIELD* instruction is a potential join point for a *get* pointcut. If such a potential join point is identified, the weaver creates a *shadow* for it. A shadow specifies the kind of pointcut to which this instruction can match. A shadow also includes the type information for the point, called the *signature*. For example, the signature of a shadow of a *GETFIELD* instruction includes the field type, its declaring type, and its name.

Finally, the weaver compares the shadow to each munger in the list. If the shadow matches any munger, the weaver implements the action embedded in the munger. For example, if the munger represents `before` advice, the weaver inserts the method body of the before advice immediately prior to the bytecode identified by the given shadow. Note that the matching process of a munger and a shadow compares the signatures of the pointcuts of the munger against the signature of the shadow. Different pointcuts specify different criteria for matching. For example, for a *get* pointcut, a match occurs when the modifier, return type, declaring type, and name of the two signatures are compatible (in terms of the inheritance hierarchy) with each other.

4.2 Creating shadows

To enable our `ifCheck`, we also modify the shadow creation process. Each kind of pointcut has a certain bytecode instruction as its potential join point. For our `ifCheck`, all conditional branch bytecode instructions are the potential join points. However, since our focus is on conditional branches for which the conditional consists of a field or local variable use, we filter the potential join points to be conditional branches that operate on stack operands produced by *GETFIELD*, *GETSTATIC*, and **LOAD* bytecode instructions.

There are two types of conditional branch instructions in the Java bytecode instruction set: Those that require one stack operand (e.g., *IFEQ*), and those that require two stack operands (e.g., *IF_ICMP*). For the first type, we need only check whether the predecessor of the bytecode instruction is a field or local variable access. For the second type of conditional branch instruction, we must check two stack operands for a match. To identify the operands, we perform reverse abstract interpretation from the matched instruction to find the producers of the instruction operands. To implement this process as efficiently as possible, we terminate this process if we reach an instruction with multiple predecessor instructions, potentially failing to identify a join point. We found however, that given the ajc compilation strategy, we are able to find all join points in the programs that we studied. In addition, we found that the scan length is commonly less than four instructions on average.

Once we determine the conditional branch instruction that we are considering is a potential join point of the `ifCheck`, we next generate an appropriate signature for the point that we will later use to match against the signature of the pointcut. For all of the current join point kinds in AspectJ, we can extract the information needed to create a signature from the instruction. For example, we can extract the type, declaring type, and name of a field directly from a *GETFIELD* instruction. Similarly, we can extract all pieces of the signature for the *INVOKE** instruction (name, declaring type, parameter types, return type, etc.) directly. For conditional branch instructions, however, there is no information embedded in the *IF** instruction. We must extract this information from the corresponding field access or local variable load instructions. We perform this step during reverse abstract interpretation.

Since there may be more than one signature for one conditional branch instruction that we must compare against the pointcut, we generate a linked list of signatures. We generate one signature for each field or local variable access. We implement the signature linked list as a subclass of the current signature class, with one extra field, called "next". We store the reference to the signature list as AspectJ does for all other signatures in a shadow object.

4.3 Matching Mungers against Shadows

We next modify the matching process of mungers and shadows in support of our new pointcut. As we describe above, the system eventually delegates the matching of mungers to shadows to matching the signature of munger pointcuts to shadow signatures. Since the signature of the shadow of a potential join point for the `ifCheck` pointcut contains a linked list of signatures, we compare the signatures in the list to the pointcut signature until we find a match. For the field access signature, the matching process is similar to the *get* pointcut: We compare the return type, declaring type, and name. If the field is not found in the current declaring type, we also examine all supertypes. For local variables, we need only compare the name and return type.

For the `ifCheck` pointcut to be used as other pointcuts, we also define before, after, and around advice for the `ifCheck`. We define `before` to be the point immediately preceding the conditional branch instruction. We define `after` to be the point immediately following the conditional branch, that is, immediately prior to the first instructions of both of the target instructions of the branch. We define `around` as both `before` and `after`.

Given this implementation, we can write:

```
before(): withincode(void hello())
    && ifcheck(* TRACE)
{
    System.out.println("before check TRACE: "
        + thisJoinPoint);
}
```

which inserts the code body before every *IF** bytecode that employs a boolean expression that uses a field or local variable with the substring "TRACE" in its name, within the "hello" method. Note that this example requires that we add extra code to the AspectJ runtime module to support the implementation of *thisJoinPoint*.toString() (to print the string above).

4.4 Declare Statement Extension

As we stated previously, AspectJ implements two types of actions: advice and declare statements. Advice guides the weaver to insert code blocks at the join points identified, while declare statements specify non-weaving actions.

We extend AspectJ with an general-purpose action, called *declare location*. This action marks and reports the source lines of the matched join points. We then exploit this action within Eclipse to implement AOP-guided folding. However, this action is also useful for debugging aspects.

The declare location is an action that is static, i.e., it cannot rely on runtime information to select join points. The format of the declare location statement is similar to declare error and declare warning and we specify it as follows:

```
declare location: pointcuts [: message]
```

message is an optional parameter that, if specified, is communicated along with the source location. All pointcuts, including the `ifCheck`, can use the declare location action. For example,

```
declare location: withincode(void hello())
    && ifcheck(* TRACE)
    : "checking TRACE in hello()";
```

marks and reports all lines of code at which a variable with the substring "TRACE" in its name, is used in a conditional branch expression within the hello() method. For each such location, a message "checking TRACE in hello()" is attached. In this example:

```
pointcut foldCatchBlocks() :
    handler(!NullPointerException);
declare location: foldCatchBlocks();
```

the pointcut identifies all catch blocks that can catch an exception other than the NullPointerException. The declare statement, causes the system to identify the locations in the code at which these points occur and report them to the IDE or standard out. In this example, we do not specify the message.

The implementation of declare location is similar yet simpler than that of the `ifCheck`. The parsing process of advice and declare statements are handled by the PatternParser. We extend this class as we do for `ifCheck`, to parse declare location statements. We create an object of a special kind of ShadowMunger, called a LocationChecker, for this statement. The object contains the parsed pointcuts and messages. During the munger and shadow matching process, the LocationChecker delegates the matching to its enclosing pointcuts. If the signature of a pointcut matches the signature of the given shadow, the LocationChecker records the source location of the join point of the shadow. The LocationChecker returns "not matched" *regardless* of whether a match is made to turn off the weaving process for the declare location action.

4.5 Exploiting Declare Location Advice in Eclipse

We can exploit the location markers generated by our declare location action in a number of ways. In this section, we present one such application in which we use the location report within an Eclipse IDE plugin that we have designed. The plugin, called *AJFolderPlugin*, consumes a location report and generates a projection (a view) of the Java source code file. The projection folds away code blocks identified by the source locations in the report.

In Eclipse, the project build process with AspectJ is handled by the plugin AJBuilder. This builder sets up the build environment, and then invokes the AspectJ compiler and weaver to do the real work. The AJBuilder keeps a list of listeners that it can alert when compilation begins and end via the *IAJBuildListener* interface methods preAJBuild and postAJBuild. We monitor and intercept this build process with the AJFolderPlugin using the IAJBuildListener interface. We use the postAJBuild method to collect the location reports produced by the builder, if any, and then update the location map for the built project.

Eclipse implements a general folding mechanism in the *ProjectionView*. The ProjectionAnnotationModel specifies folding behavior using a list of annotation entries each designating the start and end offset of the folding area. The model also indicates whether the folding area is collapsed or expanded. Each time the list of annotations is replaced, the model updates the view automatically.

We implemented the postAJBuild method to generate a new annotation list and to update the model each time a build completes. We use the location report to generate the list. As a result, all locations that the AOP system identifies for any currently open source files are folded automatically. The location report specifies the point in the source at which folding begins. This is a point in the code that either contains parenthesis or curly brackets. If an if statement does not employ curly brackets, we insert them to enable folding. The AJFolder plugin identifies the offset in the code that ends the scope started at that point in the code (a close parenthesis or curly bracket). The plugin replaces the source lines with a single line containing the string "{ . . . }" at the point at which the curly braces start, or "(. . .)" at the point at which the parenthesis start. The plugin also places a marker at the source line that the user can use to manually expand and collapse the line at any time.

The AJFolderPlugin also enables users to specify the various folding options via the Window preferences page of the Java Editor. This overrides the default folding functionality which employs an all-or-nothing approach – either all methods, comment blocks, or imports are folded or they are left expanded. Our mechanism enables selection of such blocks as well as blocks and regions *within* methods.

This implementation enables folding at a fine grain given any point in the code with parenthesis or curly braces. This includes try and catch blocks, synchronized scopes, loops, parameters, and expressions, in addition to if-then-else blocks. Other programmers and researchers have shown how to implement pointcuts for synchronized scopes [23] and loops [12]. Our `ifCheck` pointcut captures if-then and if-then-else blocks and their parenthesized expressions. The default AspectJ implementation includes support for identifying methods (and thus method signatures containing parentheses) and catch blocks (via `handler` pointcuts). We can extend (and plan to as part of future work) this latter support to include try blocks which are specified as part of the method exception attribute in the bytecode, so that they can also be candidates for folding.

In the next section, we describe the unique ways that we can select these individual code regions for *selective* collapsing or expanding of blocks. The examples below show how to specify the pointcuts for particular folding candidates.

```
pointcut foldMethod() :
    execution(public static *foo(..));
declare location: foldMethod();
declare location: ifCheck(* Options.DEBUG);
```

The first declare statement identifies all public, static, methods with the name foo (regardless of their parameter and return types), and extracts them as folding candidates. The second declare statement, identifies all if expressions in the source that use a field from the Options class with the string "DEBUG" in its name.

5. Identifying Folding Points in the Code

Our next step in the process of AOP-guided source folding, is to identify scopes in the code that should be folded to best clarify the underlying semantics or behavior of a Java source method or class definition. We describe three ways in which we do this. The first is to use a database of names that are commonly used for variables in expressions that begin if-then and if-then-else scopes of cross-cutting concerns. The second is the use of profile information to identify infrequently (and frequently) executed blocks. The third is the use of domain specific information that is specified by the programmer.

5.1 Automatic Generation of Pointcuts Using Common Variable Names

Our first technique for identifying folding points in the code is based on our observations of and experiences with open-source systems. We find that there are a small number of commonly used variable names that programmers use as part of if-expressions to implement cross-cutting concerns. The names that we employ currently are:

```
DEBUG, TRACE, LOG, MEASURE, TIME, TIMING,
SANITY, ASSERT, PRINT, OPTION, PROFILE,
CHECK, VERBOSE, DUMP, TRACK
```

Our string matching process is insensitive to case.

To ease the job pointcut implementation by users of our system, we keep a per-project database, called the autoAOPDB, of these names as part of the Eclipse IDE. When a user creates an new aspect, she can choose to have ifCheck pointcuts for these commonly used strings, automatically (re-)generated for her. The user can add and delete strings from this list at any time. Here are examples of the automatically generated declare location advice that use the ifCheck pointcuts and default string names:

```
declare location: ifCheck(* DEBUG);
declare location: ifCheck(* PROFILE);
declare location: ifCheck(* VERBOSE);
```

We also use the autoAOPDB to store names of exception handler types and method names. This enables us to generate pointcuts which fold away entire methods and catch blocks, e.g., we fold catch blocks that expect the type Exception using:

```
declare location: handler(Exception);
```

We can also extend this component to identify names automatically from the source or bytecode of a program. For example, we can identify variable names that have boolean types or boolean fields that are static and final. We plan to investigate such techniques as part of future work.

One limitation of our system (that results from the use of the Java compiler employed by Eclipse), is that the compiler performs some optimization that breaks the connection between the source and the bytecode in some cases. One example of this, is that the compiler eliminates if-blocks if it can determine that the condition is always false, e.g., the conditions checks a boolean static final field that is set to false. To avoid this during program development or investigation, we must turn off such optimizations to enable folding. We can then enable optimizations for final or release builds of the system.

5.2 Profile-Guided Selection of Folding Pointcuts

Another unique way in which we can automatically specify and employ AOP-based folding, is to use profile information. A profile consists of information about the execution of a program. Profiles are commonly used to guide optimization (by hand or automatic) and can give programmers insight into how their code is exercised by an input. We automate this latter process by hiding (or exposing) infrequently executed blocks.

During the build process within Eclipse, we modify the bytecode that the system compiles during the bytecode analysis step of AOP weaving. Eclipse employs the bytecode engineering library (BCEL) to manipulate, analyze, and optimize the bytecode. We extend this process to insert profile collection instructions into the bytecode. These instructions count the number of times each instruction (or basic block) is executed.

They type of profiling that we can perform is flexible. We can insert instructions that count the number of times the fall-through and branch target instructions are executed. We use this information to determine whether the fall-through or the branch is taken more frequently. The resulting profile contains the bytecode index of each conditional branch instruction with 1 bit of information indicating that the fall through (0) or the jump target (1) is taken more frequently. Currently, we do a simple compare as to which is greater and mark the fall through if they are equal. As part of future work, we plan to investigate ways of having the user set a threshold that determines which code regions should be marked.

During aspect generation, a user can select to incorporate this profile information. To do so, the user specifies an input to the program (as part of the current Run process). At the end of the build process, we execute an instrumented version of the program in the background to collect the profile data. When a profile is available, the user presses a button to incorporate the information into the folding aspect. The user also specifies whether she is interested in folding away infrequently used blocks or frequently used blocks. We use the former to expose commonly executed code and potential optimization opportunities. For example, we can use it to identify code regions that are, perhaps unexpectedly, commonly executed as well as when, which, and how commonly used data structures are accessed. Folding unused code regions can reveal whether the input is an appropriate coverage test, how to generate such tests, and to identify untested code regions that may be hiding bugs.

In addition, we can change how Eclipse exploits the AOP-generated code markers when the users employs profile-guided pointcuts. Instead of folding code, we can highlight frequently executed regions. We can employ any type of source-line-based visual technique that is available as a plugin in the same way as we do for the AJFolderPlugin. This is useful if we are interested in other types of profiles, e.g., field accesses. We use the profile data to identify the most common field accesses. Eclipse can then use the markers at these points to highlight the corresponding source instructions.

To implement AOP-based folding using profile data, we read in the profile data and store the bytecode instruction line numbers as part the autoAOPDB. We add the bytecode indices as strings and during pointcut parsing, we check whether the name starts with an integer (which is not allowed in Java for field, method, and variable names). If it does, the matching process simply checks whether the bytecode index is the same as the one in the pointcut and is an IF* instruction. If it is, then the instruction is marked as described previously and communicated to the execution environment with a message set to be the bit of information indicating fall through or branch target. During folding (or highlighting), we use this information to fold the if-part of the the source or the else-part of the source. If there is no else-part and the fall through is selected, we fold the if block, otherwise we perform no folding. Note that we need not incorporate the profile data into the autoAOPDB but instead use it within Eclipse directly. We do so to maintain an uniform interface between aspect generation and application and the IDE and so that the user has control over what is folded and what is not (via a view within the IDE) throughout the lifetime of the project.

5.3 Domain-Specific Selection of Folding Pointcuts

The final technique we incorporate for selecting folding pointcuts uses domain specific string names. Commonly, users employ intuitive method names that indicate the functionality implemented by the method. For example, in JikesRVM's garbage collection system, methods related to allocation contain the substring `alloc` in their name.

We can automatically fold source code according to these domain substrings using our system. The user can specify a folding pointcut such as:

```
declare location: !execution(* *ALLOC(..));
```

A user can add these strings to the autoAOPDB. The execution pointcut identifies all methods with the type signature indicated – this is original the AspectJ behavior. The declare location causes the method to me marked as opposed to weaving in any code. The marking is then used by the folding system to expand or hide code. If !execution is used, then all methods *except* those with "ALLOC"

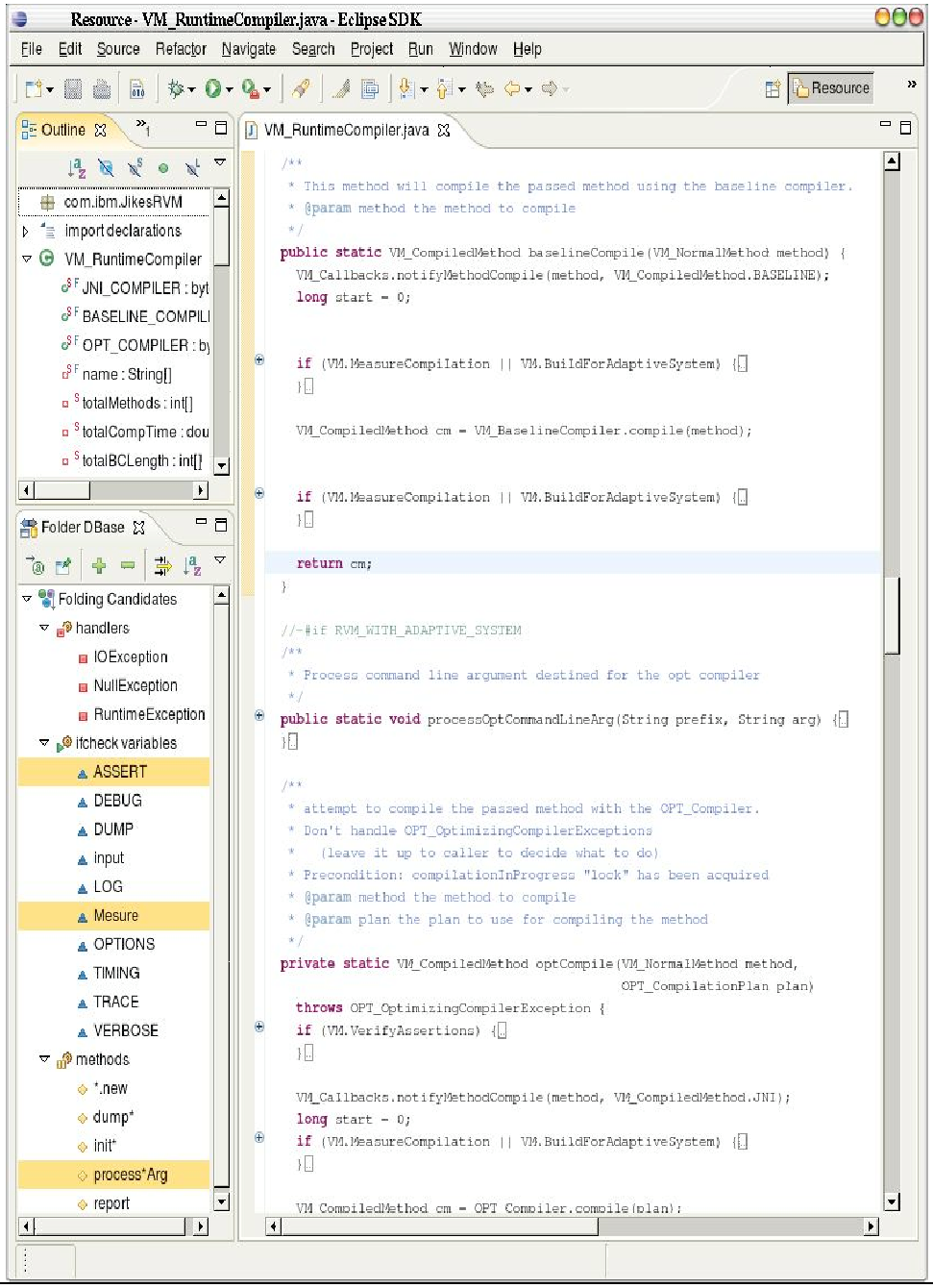

Figure 2. AOP-Guided folding within the Eclipse framework.

Program	Description
JEdit	Visual text editor
Jetty	HTTP servlet server
JikesRVM	Java Virtual Machine with adaptive optimization and multiple GC systems
Joeq	Research Java Virtual Machine
SpecJBB	SPEC transaction processing application
Weka	Machine learning algorithms for data mining

Table 1. Benchmark descriptions for the open-source software systems that we analyzed.

Benchmarks	Count	Total LOC	Average Method LOC	Methods with Folding Opportunities	
				Count	Percent
JEdit	552	38267	69.45	213	38.59
Jetty	149	11193	75.63	71	47.65
JikesRVM VM	135	12094	89.59	64	47.41
Joeq	175	18737	107.68	91	52.00
SpecJBB	162	11704	72.70	127	78.40
Weka	784	60016	76.55	287	36.61
Average	326	25335	81.93	142	43.59

Figure 3. Benchmark characteristics for our set of large, open-source, programs

in their name will be folded. We modify exact matching of names to enable substring matching.

In Figure 2 we show a snapshot of our system and a folded JikesRVM file. The file is from the compiler harness and is called VM_RuntimeCompiler.java. This is the file from which we extracted the code in our first example, Figure 1. The autoAOPDB in the Folder DBase view on the bottom left. These are the string names in the database that we employ for automatic pointcut generation. We have employed a number of different declare location actions for this file: The if-statements are folded via the `ifCheck` pointcut and the method is folded using the execution pointcut. The latter employs our domain-specific pointcut generation which we specify as:

```
declare location: !execution(* *COMPILE(..));
```

We have chosen not to fold the parenthesis so that the reader is able to see which names were matched by the autoAOPDB-generated pointcuts, e.g., measure and assert.

6. Evaluation

In addition to the visual snapshots that we present in the previous section that show the impact of our system, we also evaluate our system using a number of different benchmarks. In the subsections that follow, we describe our methodology and then present our empirical findings.

6.1 Empirical Methodology

We implemented our system as plugins as described in the previous sections to the Eclipse framework version 3.1 using the Java 2 Platform version 1.4. Eclipse implements AOP via AspectJ plugins. Our system uses the AspectJ Development Tools version 1.3.

To evaluate our system, we selected two sets of benchmarks. The first set includes large applications from the top Java downloads (with source freely available) from SourceForge.net [21], the Jikes Research Virtual Machine [2], and SpecJBB from the Standard Performance Evaluation Corporation (SPEC). We list these programs in Table 1, and briefly describe what they do.

Benchmarks	Count	Total LOC	Average Method LOC
compress	3	111	37.00
jess	22	1434	62.35
mtrt	15	59	871.00
Average	13	535	323.449

Figure 4. Benchmark characteristics for a subset of the SpecJVM98 benchmarks

We present the overall source characteristics of these benchmarks in the table in Figure 3. The second column is the number of methods that we processed from each benchmark. We only considered methods with 30 or more lines of source code. We specify the total number of source lines that we processed in these methods in column 3. The fourth column is the average lines of code per method. The fifth column is the number of methods in each project that contained at least one opportunity for folding (and if statement with one of our autoAOPDB strings in the expression). The final column shows the percentage of these methods given all of the methods that we analyzed.

On average 44% of the methods that we analyzed had implemented folding opportunities, i.e., cross-cutting concerns guarded by if-statements. The average number of source lines in the methods that we analyzed was 82.

We also evaluate the efficacy of profile-guided selection of if-statements. Such selection may be useful to users when source code does not contain any obvious key words that are used in if-statement conditional expressions. Folding of frequently or infrequently used code blocks can also be useful to programmers for identifying bugs and optimization opportunities. For this study, we employed the a subset of the SpecJVM98 benchmark suite [22]. We include the programs from this suite for which source was available. We present the source characteristics for these programs in the table in Figure 4.

6.2 Results

To evaluate the efficacy of our system in terms of the number of source lines eliminated from view as a result of AOP-guided folding, we analyzed our large-program benchmark set. We counted the total number of source lines in methods with greater than 30 source lines. We selected 30 as our threshold since that is the number of lines that is viewable using the default Eclipse settings on the author's laptop.

We also counted the number of lines that remained after we performed the folding. We employed autoAOPDB selection using the list of keywords that we specify in Subsection 5. We also employ folding for catch blocks as we describe at the end of Subsection 4.4. The declare statement and pointcut that we use to enable this is

```
pointcut foldCatchBlocks() :
    handler(!NullPointerException);
declare location: handler(*);
```

Figure 5 shows the percent reduction in the number of source lines in methods for which there is a folding opportunity. The first bar for each program shows the percent reduction enabled by both the `ifCheck` pointcut and catch block folding. On average, we eliminate 13% of the source lines from view. The second bar is the percent reduction do to the `ifCheck` pointcut and autoAOPDB selection alone (without catch block folding). On average, we eliminate 8% of the source lines.

Most programs achieve a majority of the reduction from the `ifCheck` pointcut and autoAOPDB selection. JEdit and Jetty

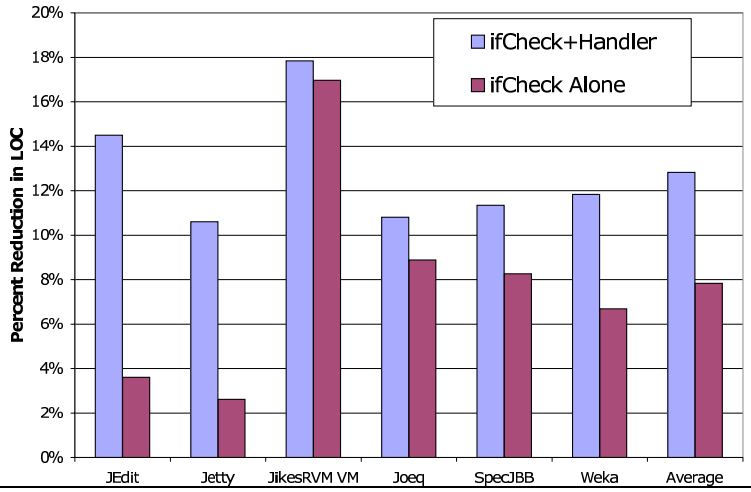

Figure 5. Percent reduction in source lines when we employ autoAOPDB generation of pointcuts (Subsection 5) and AOP-guided folding.

show reductions primarily as a result of catch block folding. These programs contain very few cross-cutting concerns that are guarded by if-statements. For the methods that do contain such code, the primary keyword matched is "DEBUG". "DEBUG" is the most popular keyword across all programs, with "TRACE" and "VERBOSE" following second.

We also investigated the efficacy of using profile information to guide automatic generation of aspects. For this study, we counted the number of times the basic block targets of each conditional branch were executed for a specific input. Our profile identified the conditional branch and whether or not the fall through or the branch target basic block was taken less often. We then used this profile to generate pointcuts for each conditional branch. The string name filter in each pointcut is the bytecode index. The declare action marks these source lines and Eclipses uses the markers to perform folding.

Figure 6 shows the results for a subset of the SpecJVM98 benchmarks. We only employed those benchmarks for which the source is available. The compress benchmark benefits the most for profile guided folding since many of the same code blocks in the program are repeatedly executed; we fold away the others. On average, we reduce the number of lines of code that the user sees by 9% using this technique.

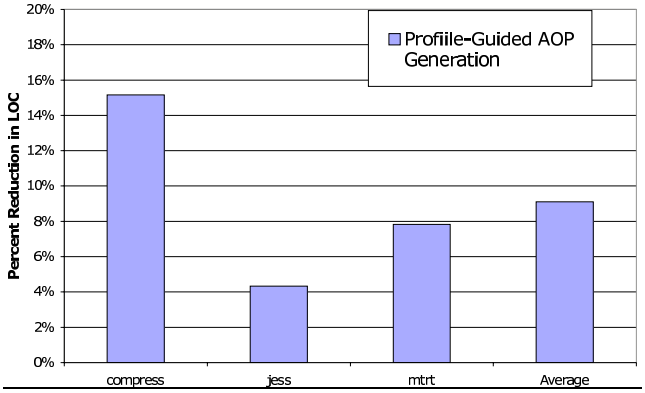

Figure 6. Percent reduction in source lines when we employ profile-guided generation of pointcuts (Subsection 5.2) and AOP-guided folding.

7. Related Work

Our system is unique in that it combines and extends techniques from AOP, refactoring, and profile guided optimization for the purpose of improving source code clarity and understanding. There has been a significant amount of related work in these areas but none that, to our knowledge, combines the techniques and employs the resulting system in the same way.

The research most related to the our work is aspect mining. The authors in [7] survey current aspect mining and refactoring techniques and motivate the need for AOP refactoring in legacy systems. Aspect Browser[9], AMT [10], and AMTex [24] each exploit naming conventions to identify cross cutting concerns in legacy code. The latter two tools combine string names with type information. Shepherd et. al in [1] employ and extend a program dependence graph approach for clone identification with which they couple data flow information to refine and extend the mining process without help from the user.

In our system, we use a very simple, ad-hoc approach to mining that relies on common programmer idioms or on the programmer herself to specify string names of class members. We store these names in a database and generate `ifCheck` pointcuts from them automatically. The programmer can add or delete names from this database. In addition, we employ profile information to identify code behaviors that are useful to fold. Since our system identifies these concerns each time compilation (incremental or batched) is invoked, identification of these concerns must be fast, precluding resource-intensive approaches of extant techniques.

Our work is also related to research that describe AOP extensions, in particular, research that implements new join points. The goal of our join point extensions however, are different from all extant join points in that they are not intended to modify the dynamic behavior of a program. Instead they introduce markers into the bytecode which a source-level visualization tool such as Eclipse can use to fold away unrelated concerns or to highlight important source blocks (i.e. those commonly executed during profiling).

Two join points that are most similar to ours is the implementation of basic blocks join points described in [20] and the discussion on supporting if-then-else join points in [12]. The former work extends AspectJ with conditional and iteration pointcut designators and expressions to enable code testers to express test adequacy analysis relative to cross cutting concerns. In [12], the authors present the design and implementation of a loop join point which they use to specify loop-based parallelism. They discuss the

potential implementation of extending their system to model an if-then-else join point.

Folding is another mechanism which we incorporate within our framework. Folding is a technique used in Eclipse to hide lines of the source program from view. Currently, Eclipse implements a folding plugin to fold all comments, header comments, entire method bodies, import lists, and inner classes. We implemented a new folding plugin that folds any code body within curly braces and parentheses. We then link this functionality to the extended AOP plugin which identifies points in the code at which folding should occur. As such, we perform folding at a finer grain, i.e., the default folding applies folding to all methods when selected and we apply folding to individual methods that do not match the domain of interest.

8. Conclusions

To improve code understanding *without* requiring modification of the program, we present a set of techniques that employ and extend AOP and the Eclipse integrated development environment. Our system hides from view unrelated code bodies within the scope of conditional branches, methods, and catch blocks. We employ new pointcuts and filters to identify these program entities and a new action that marks bytecode instructions for use by a source code visualization system, such as the Eclipse IDE. We extend Eclipse to consume these markers and use them to guide code block folding and highlighting.

We automate generation of the aspects by identifying branches that use common and domain-specific string names within their expressions, e.g., DEBUG, VERBOSE, MEASURE, ALLOC, etc. We also show how to couple profile information with automatic aspect generation to enable marking of frequently (or infrequently) used code blocks.

Our empirical evaluation of the system indicates that 44% of all methods in popular, large-scale Java software systems implement cross-cutting concerns using if statements that are guarded by a small number of intuitive keywords. Our system reduces the number of source lines by 8% on average for methods that have an average size of 82 source lines. We also show that we are able to avoid 9% of source line by identifying via profiling unused conditional branch bodies and folding them from view.

References

[1] D. S. abd Jeffrey Palm, L. Pollock, and M. Chu-Carroll. Timna: A framework for automatically combining aspect mining analyses. In *International Conference on Automated Software Engineering*, November 2005.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. de Moore, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *International conference on Aspect-oriented software development (AOSD)*, Mar. 2005.

[4] S. Breu and J. Krinke. Aspect mining using event traces. In *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 310–315, 2004.

[5] S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *International conference on Aspect-oriented software development (AOSD)*, pages 102–111, 2004.

[6] L. Cole and P. Borba. Deriving refactorings for aspectj. In *International conference on Aspect-oriented software development (AOSD)*, Mar. 2005.

[7] A. v. Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03).*, Nov. 2003.

[8] Eclipse.org. Eclipse framework, 2001. `http://www.eclipse.org/`.

[9] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Workshop on MDSOC*, 2000.

[10] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE) 2001*, 2001.

[11] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *International conference on Aspect-oriented software development (AOSD)*, Mar. 2005.

[12] B. Harbulot and J. R. Gurd. A join point for loops in aspectj. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, 2005.

[13] J. Hugunin. Guide for developers of the aspectj compiler and weave, 2004. http://dev.eclipse.org/viewcvs/index.cgi/ checkout /org.aspectj/modules/docs/developer/compiler-weaver/index.html?rev=1.1&content-type=text/html&cvsroot=Technology_Project.

[14] G. Kiczales. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[15] C. Krintz. Using adaptive optimization techniques to teach mobile java computing. In *Workshop on Intermediate Representation Engineering for Virtual Machines (IRE)*, pages 41–46, 2002.

[16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, Apr. 1999.

[17] M. Marin, L. Moonen, and A. van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *Workshop on Modeling and analysis of concerns in software*, pages 1–5, 2005.

[18] D. Nelson and Y. M. Ng. Teaching computer networking using open source software. In *Conference on Innovation and technology in computer science education*, pages 13–16, 2000.

[19] J. Nieh and C. Vaill. Experiences teaching operating systems using virtual platforms and linux. In *Technical symposium on Computer science education*, pages 520–524, 2005.

[20] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.

[21] Sourceforge.net. http://sourceforge.net/.

[22] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[23] Synchronized Block Pointcut Semantics. `http://blogs.codehaus.org/people/jboner/archives/001134_semantics_for_a_syn%chronized_block_join_point.html`.

[24] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139, 2003.