Ophir:A Framework for Automatic Mining and Refactoring of Aspects

Technical Report No. 2004-03

David Shepherd and Lori Pollock

Department of Computer & Information Sciences
University of Delaware
Newark, DE 19716

28th October 2003

Ophir:A Framework for Automatic Mining and Refactoring of Aspects

David Shepherd and Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{shepherd, pollock}@cis.udel.edu

1. INTRODUCTION

Aspect oriented programming (AOP) enables programmers to organize their program in a way that makes it easy for another programmer to understand not only the dominant organization (e.g., classes in object oriented programs), but also the programming concerns that would have appeared scattered if they were factored according to the dominant decomposition[1], [2], [3]. Example simple applications in which aspect oriented programming can provide a fast and easy-to-understand implementation without inserting code snippets into numerous unrelated methods include instrumenting specific events to provide logging during a program's execution and managing display updates in a figure editor [4].

Typically, programmers decide to exploit AOP when they are implementing a new task in a program that, because of its nature, is going to require adding code in locations scattered throughout many functions, classes, or files. However, the debugging, testing, and maintenance of large legacy systems also could be eased considerably if already existing code for these kinds of tasks could be identified and refactored into AOP style. Even after refactoring, any project where there are multiple programmers, especially if geographically separated (such as an open source project), could benefit from ongoing support for identification of aspects. Programmers are unlikely to be completely familiar with other programmers' code, causing them to miss opportunities for applying aspects to their problems.

Transformation to an aspect oriented program requires two steps: (1) *mining*, or identification, of code to perform a single task scattered throughout a software system, (We call the task a *concern* or *refactoring candidate*.) and (2) refactoring of the legacy system into an aspect oriented program in which the scattered code has been replaced by aspects in an AOP language such as AspectJ[4]. A lightweight approach to aspect mining is to perform lexical searches, which can be combined with type information[5], [6]. This approach is effective in finding some concerns very quickly. However, more useful lexical searches are dependent on the coding practices of the programmer, such as variable or method naming conventions, which are hard to guarantee, especially in a legacy system[5], [6]. More sophisticated tools have been developed that are related to, but not explicitly built for aspect mining. JQuery[7] and FEAT[8] both incorporate semantic information to navigate the source code. They focus on providing intelligent exploratory capabilities, with the user controlling much of the function.

The tools currently available for aspect mining require a large amount of user input and/or knowledge of the program. The lexical tools require a regular expression or string be given as a search target as well as the intelligence to sort through the returned data. The exploration tools require a user to develop a high level understanding of the program to use the tool to navigate around the program. Currently, refactoring must be done manually, which seems to conflict with some AOP goals, which include on-demand remodularization[3].

To address the problem of automating the aspect mining and refactoring processes, we have designed and implemented Ophir, a framework to support automatic mining analyses and manual or automatic refactoring of aspects as desired. In this paper, we describe the design and implementation of Ophir, Ophir's use in aspect mining, the Eclipse-specific features exploited in its implementation, and how Ophir, with its base in Eclipse technology, can be used as a springboard for automatic aspect mining and refactoring research and tool development.

2. OPHIR

From our experience using lexical tools and FEAT for aspect browsing and mining, we have learned that, due to the non-uniform nature of concerns, different kinds of analyses for aspect mining are needed to capture the variations. Ophir currently includes a mining analysis that is more substantial than previous mining analyses (i.e., lexical) and is automated (i.e., requires no user input). In addition, Ophir's API allows others to easily add new analyses via Eclipse's plug-in model, displaying their results in Ophir's Perspective. Because of its close ties to the underlying JDT plug-in, the design of our framework also promotes future work in automatically refactoring the discovered concerns [9]. The next subsections describe the aspect mining problem in more depth, the analyses currently within Ophir for aspect mining, and the use of Ophir.

2.1. The Aspect Mining Problem

Mining aspects is difficult because, at first look, these aspects seem not to follow any pattern or structure. If viewed on a device like AJDT's visualizer[9], some concerns seem to be hopelessly scattered throughout the classes. However, as we look closely at these concerns, patterns emerge.

For instance, if we start mining with a lexical tool, we might try to search for code that displays different objects on the screen seeding the search with the string "display". In the ideal case, the word "display" would have been used as the title to a method in each class that was used to display that object. Then, the "display" concern would be an example of a perfect crosscutting pattern, as it would have one method in every class, cutting a new dimension across the object-oriented modularity. However, if the word "display" had been used in any other manner, the lexical tool would also find these instances of the word display, returning extra noise among the helpful results. Furthermore, if the word "showIt" had been used instead of the word display for the method names in a few classes, our search for "display" would have only returned part of the concern.

If this first search was not helpful, we might try another seed. In order to find a good seed, assume that we have already located some code that is part of the display concern. Furthermore, assume that part of this code is present in most, if not all, of the other classes' display code. Can we find the other classes' display code using this as the seed? Most likely not, even if other classes include this code (or very similar code) in their display code. Lexical searching will fail if the code has been reordered without changing the semantics, whitespace has been added, variable names have been changed, or there has been code inserted between lines of the code we are searching for. In general, concerns are a high level concept that have no particular syntactic properties when translated into code.

2.2. Mining via IRs

Our automatic analysis for aspect mining successfully overcomes many of the shortcomings of lexical analyses, and is currently implemented for one of the most popular patterns of concerns. The analysis is based on AspectJ, arguably the most used AOP language available (just based on the number of books on AspectJ)[4]. While it may not address all of the larger goals of the AOP community, it addresses many of them. Future AOP languages are almost certain to share many of the basic principles of AspectJ, and so analyses on AspectJ can serve as models for analyses of other languages[4].

In AspectJ, there are several types of *advice*, such as: "before", "after", and "around". This advice (code) can be executed at a specified joinpoint. *Joinpoints* are the points that one can specify (using AspectJ) within a program to execute a code segment, points such as the beginning of a method or before an access to a field. In our work with AspectJ, we noticed that one of the most popular forms of advice is "before" advice that executes before a method in a specified set of methods is run. This type of advice is the target of our analysis.

We search for "before" advice using intermediate program representations (IRs) and code clone identification technology. When we find clones, they are often code that should have been written as an aspect. A similar analysis could be performed to find "after" advice and even "around" advice.

We use two program representations to help us mine aspects. Based on Komondoor and Horwitz algorithm for identifying duplicate code segments in source code[14], the Program Dependence Graph (PDG)[13] representation of each method helps us identify clones of the control and data dependence structure in different methods. In a PDG representation of a method, each node represents a statement, and an edge between two nodes represents either a control or data dependence relation between the corresponding statements. We exploit the key prop-

```
public void write(StorableOutput dw) {
   super.write(dw);
   dw.writeInt(fOffsetX);
   dw.writeInt(fOffsetY);
   dw.writeStorable(fBase); }
```

```
public void write(StorableOutput dw) {
   super.write(dw);
   dw.writeStorable(fLocator);
}
```

(a) Calls to “super.write” and “writeStorable” could be moved to an aspect as “before" advice.

```
public void figureInvalidated(FigureChangeEvent e) {
   if (fListeners != null) {
      for (int i = 0; i < fListeners.size(); i++) {
         DrawingChangeListener l = (DrawingChangeListener)fListeners.elementAt(i);
         l.drawingInvalidated(new DrawingChangeEvent(this, e.getInvalidatedRectangle())); }}}
public void figureRequestUpdate(FigureChangeEvent e) {
   if (fListeners != null) {
      for (int i = 0; i < fListeners.size(); i++) {
         DrawingChangeListener l = (DrawingChangeListener)fListeners.elementAt(i);
         l.drawingRequestUpdate(new DrawingChangeEvent(this, null)); }}}
```

(b) All code except calls to “drawingRequestUpdate” and “drawingInvalidated” could be moved to an aspect as “around" advice.

Fig. 1. Examples of Mined Refactoring Candidates

erty of a PDG that only necessary orderings due to dependences are represented. We use the Abstract Syntax Tree (AST) to determine if individual statements are similar, much like Krinke in his technique for identifying similar code[15].

Briefly, the aspect mining analysis proceeds as follows. A PDG is constructed for each method in the program, using FLEX, a compiler infrastructure for Java[16]. A comparison algorithm is performed to compare each method’s PDG with other methods’ PDG’s in order to identify clones. Given a starting node in a PDG, the comparison algorithm traverses the PDG, adding nodes to the clone result as it finds matching nodes. Since each node in a PDG maps to a statement in the program, nodes are compared for equality using the AST, providing a stronger comparison than lexical analysis.

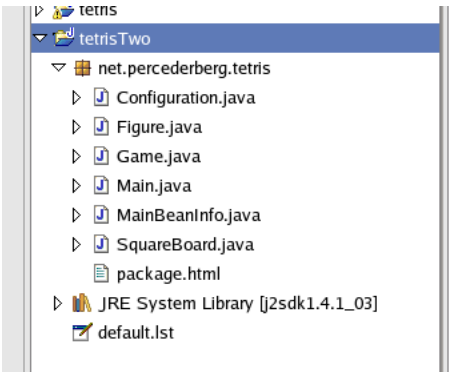
By using both IRs, our method detects clones that a lexically based clone detection system could not detect. This is because the PDG allows us to detect clones containing statements that are reordered and clones with extra statements intermingled between the cloned statements. The AST comparison allows us to detect statements that are the same except for variable name changes, constant value changes, and syntactic changes, such as adding whitespace.

The remarkable improvement in ability to detect meaningful clones becomes apparent as one examines Figure 1. Neither of these examples could have been detected by lexical means. Also, by using the IRs, we have the advantage of not requiring a seed, as lexical and exploration tools require. Also, we do not require the user to make any intelligent choices, whereas exploration tools require the user to make most of the choices and deduce most of the results.

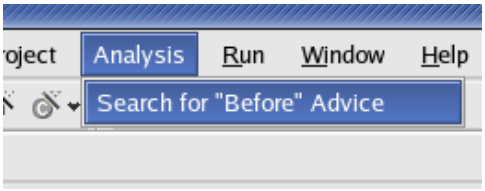
2.3. Using Ophir

1) Choosing a Project: First, a user must choose a project to analyze by selecting the project in Eclipse’s “Package Explorer” view. In Figure 2(a), we chose the project named “tetris Two”.

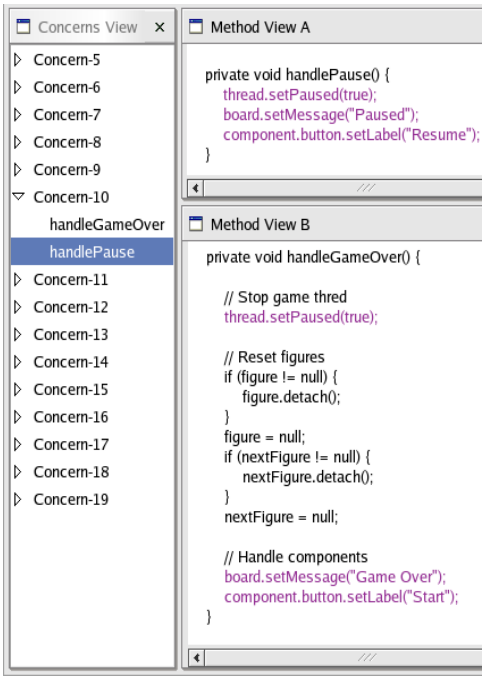
2) Starting the Mining Analysis: Next, to start the analysis, the user simply chooses the “Analysis” menu from the main Eclipse menu-bar, and then chooses the option “Search for ‘Before’ Advice”(as shown in Figure 2(b)). The analysis for aspect mining will then execute. When analysis is complete, the system will automatically change to the Ophir Perspective so that the user can view the results.



(a) Select the Project



(b) Run the Analysis



(c) View the Results

Fig. 2. Ophir With Eclipse

3) Viewing the Mining Results and Refactoring: The Ophir Perspective provides three views. The Concern View allows the user to browse each concern that has been identified by the analysis, and to see which methods contain code in that concern. If the user selects a method in a concern, that method's code is shown in Method View A, with source code lines that are part of the concern in red. If the user then selects another method in the same concern, its code will be shown in Method View A, moving the former method to Method View B.

In the example in Figure 2(c), the analysis has returned 19 concerns (in 30 secs). Prior to this display, Concern-10's method "handleGameOver" was selected, then "handlePause" was selected. Thus, "handlePause"'s source code is displayed in Method View A, and "handleGameOver"'s code is displayed in Method View B. From this example, one can see that it is likely that thread management (when to pause and unpause the thread) is probably a good candidate for an aspect, as well as message and label management. Refactoring, although not yet automated in our current implementation, could easily be accomplished by switching to the Java Perspective and using the Java Editor to edit the files.

3. Exploiting Eclipse for Implementation

Eclipse has allowed us, because of its existing plug-ins and its extensible architecture, to easily build Ophir. We have taken advantage of the following Eclipse features: (1) extension points, (2) plug-in independence, (3) existing plug-ins, (4) core plug-ins, (5) GUI (Content Provider, Menu Provider, etc), and (6) refactoring environment.

Ophir was developed as one primary Eclipse plug-in with the intent of analysis and automatic refactoring plug-ins interacting with its API. The main plug-in maintains the list of concerns (and their contents) and provides a means of displaying the concerns (the Ophir Perspective), as in Figure 3.

In order to create a plug-in that can integrate with our main plug-in, one must simply make the new plug-in require the presence of our main plug-in by editing the "plug-in.xml" file. Thus, it should be easy for anyone to write a plug-in for Eclipse that performs their analysis and extracts concerns

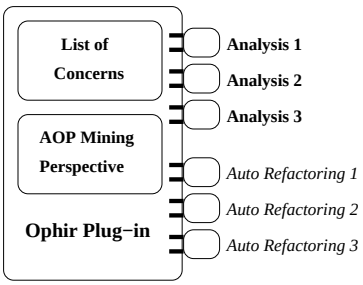


Fig. 3. Plug-in Design

into the general framework. Automatically, they are able to view their results, and even compare two methods visually. Also, we can collect concerns from many different analysis, performing a post extraction analysis on the total list of concerns, combining concerns where possible.

Eclipse already provides a powerful searching tool that almost subsumes the previous aspect mining technology (which is primarily lexical searching). By writing a simple extension point to the Search Context Menu, we can allow users to use the searching power of Eclipse to find methods, and to add methods to a concern via a simple right click.

Because we have tied the concerns' representation tightly to Eclipse's JDT Java representation, refactoring can be done without dependence on the particular mining analyses. An entirely separate plug-in could be written to move code from the concern representation to an AspectJ representation. Since the results from many different analyses will be representing concerns in this same way, the work done to refactor from the concern IR [1] will be reused for all analyses.

4. Ophir as a Springboard

We have created a framework for automatic aspect mining and refactoring which easily accommodates new analyses and refactoring techniques, and reaps the advantages of a common concern IR tied to Eclipse's underlying JDT and shared among analyses and refactoring techniques. Ophir also includes a powerful analysis that shows how semantic information embedded in intermediate program representations can be used to perform intelligent aspect mining. Due to the modularization that Eclipse provides with its plug-in framework, many universities could write analysis or refactoring tools that use our framework without our involvement. Our future work includes developing more sophisticated mining analyses, experimental comparison of concerns discovered by different analyses, and implementation of support for automatic refactoring of concerns discovered by IR-based mining.

[1]We call the concern representation the concern IR because it serves as an intermediate representation between scattered code and final form in AspectJ.

5. Biographies

David C. Shepherd is a second year graduate student in CIS at the University of Delaware. He received his B.S. in CS from Virginia Commonwealth University in Richmond, VA.

Lori L. Pollock is an Associate Professor in CIS at University of Delaware. She earned her Ph.D. in CS at University of Pittsburgh in 1986. She has served as Vice Chair and Secretary/Treasurer of ACM SIGPLAN and is a member of CRA-W.

References

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect oriented programming," in *European Conference on Object Oriented Programming*, 1997.

[2] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton, "N degrees of separation: Multi-dimensional separation of concerns," in *International Conference on Software Engineering*, 1999.

[3] Mark C. Chu-Carroll and Sara Sprenkle, "Software configuration management as a mechanism for multidimensional separation of concerns," in *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE)*, 2000.

[4] AspectJ Homepage, "<http://www.aspectj.org>," 2003, (August 26, 2003).

[5] J. Hannemann and G. Kiczales, "Overcoming the prevalent decomposition of legacy code," in *Workshop on Advances Separation of Concerns at the International Conference on Software Engineering*, 2001.

[6] William Griswold, Yoshikiyo Kato, and Jimmy Yuan, "Aspect browser: Tool support for managing dispersed aspect," in *Workshop on Multi-Dimensional Separation of Concerns at Object Oriented Programming, Systems, Languages, and Applications*, 1999.

[7] Kris De Volder and Doug Janzen, "Navigating and querying code without getting lost," in *Aspect Oriented Software Design*, 2003.

[8] Martin P. Robillard and Gail C. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies," in *International Conference on Software Engineering*, 2002.

[9] Eclipse Homepage, "<http://www.eclipse.org>," 2003, (August 1, 2003).

[10] AJDT's Homepage, "<http://eclipse.org/ajdt/>," 2003, (August 28, 2003).

[11] Amazon.com, "<http://www.amazon.com>," 2003, (August 27, 2003).

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, "An overview of aspectj," in *European Conference on Object Oriented Programming*, 2001.

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The program dependence graph and its uses in optimization," in *ACM Transactions on Programming Languages and Systems*, July 1987.

[14] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *In Proceedings of the 8th International Symposium on Static Analysis*, July 2001.

[15] Jens Krinke, "Identifying similar code with program dependence graphs," in *Eight Working Conference On Reverse Engineering*, October 2001.

[16] FLEX's Homepage, "<http://www.flex-compiler.csail.mit.edu/harpoon/>," 2003, (August 27, 2003).