Design and Evaluation of an Automated Aspect Mining Tool

David Shepherd, Emily Gibson, and Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
302 831 1953, 302 831 8458
{shepherd, gibson, pollock}@cis.udel.edu

***Abstract*— Attention to aspect oriented programming (AOP) is rapidly growing as its benefits in large software system development and maintenance are increasingly recognized. However, existing large software systems, which could benefit most from refactoring into AOP, still remain unchanged in practice, due to the high cost of the refactoring. Automatic identification and extraction of aspects would not only enable migration of legacy systems to AOP, but also prevent current systems from accumulating scattered and duplicated code. In this paper, we present the design, implementation, and evaluation of an aspect mining analysis, which automatically identifies desirable candidates for refactoring into AOP, without requiring input from the user or predefined queries. By exploiting the program dependence graph and abstract syntax tree representations of a program, our analysis is able to automatically identify a much larger set of valuable refactoring candidates than current aspect mining techniques, as demonstrated by an empirical evaluation of our automatic mining analysis on two large software systems.**
***Keywords* - Aspect Oriented Programming, analysis**

I. Introduction

Aspect Oriented Programming (AOP) is used to reduce complexity, increase readability, and improve modularity in software systems. In large software systems, complexity, readability, and modularity remain major obstacles[11], [17]. These systems, which could benefit the most from refactoring into AOP, are the last systems that are actually refactored, due to the amount of time and effort that would be required. 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[11], [17]. 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, segments of related code that should be refactored into AspectJ to provide better program readability and modularity. Programmers are unlikely to be completely familiar with other programmers' code, causing them to miss opportunities for applying aspects to their problems.

A tool that automates a large part of the refactoring process is necessary if refactoring systems for AOP are ever to become practically viable. Transformation to an aspect oriented program requires two steps: (1) *mining*, or identification, of code that performs a single task scattered throughout a software system, and (2) refactoring of the system into an aspect oriented program in which the scattered code has been replaced by aspects in an AOP language such as AspectJ[2]. This paper focuses on automating the first step of this process, the identification of *refactoring candidates*.

Existing implemented techniques and tools for aspect mining can be categorized as being either lexical or exploratory. While these approaches provide for a means to mine aspects from legacy software, neither of these approaches are automatic. Both require a seed and depend on the user's understanding of the software to be refactored.

The first kind of analysis used to perform aspect mining was a lightweight approach that performs lexical searches (a simple "Find" operation on a given text with a string or regular expression as the seed), which can be combined with type information[9], [8]. This approach is effective in finding some aspects 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[9], [8]. Lexical analysis is usually initiated by the user specifying a seed (either a regular expression or a string), and then the tool performs a grep-like function on the code base. Lexical analysis for AOP is often used in conjunction with a visualization tool, which displays the source files with the results of the search highlighted. This is especially helpful to prospective AOP programmers by letting them see the scattered nature of certain aspects. These lexical tools have been demonstrated to be most useful in situations where programmers follow a set of principles when naming their variables and methods [9], [8]. Lexical tools are now used in combination with type information to provide more intelligent mining. Execution of the analysis is still quick, yet the tool is able to find several kinds of aspects that lexical analysis alone cannot discover. Some systems even rank the amount of scatter that different types in a program have in order to decide which code segments should be part of an aspect[19].

Exploratory tools have been developed that are related to, but not explicitly built for aspect mining. JQuery[18] and FEAT[16] both incorporate semantic information to navigate the source code. They focus on providing intelligent exploratory capabilities, with the user controlling much of the function, in order to lead the user to the discovery of an aspect. These tools specialize in giving the user ways to navigate more quickly and intelligently around code to eliminate many fruitless searches.

While the tools currently available for aspect mining provide support for the process of identification of aspects in existing software, each approach suffers from one or more of the following drawbacks:

- Tool users are required to have a considerable amount of knowledge about the overall structure and function of the program being analyzed.
- The programmer must specify a search seed as input to the analysis.
- The identification and filtering of potential refactoring candidates is not fully automatic, requiring considerable time by the tool user to eventually lead to aspect identification. While full automation is a lofty goal, simple aspects can be identified automatically.
- The identification analysis can miss desirable aspects.

The main downfall of lexical searches is the requirement of a user to input a seed. The formulation of a seed that will return meaningful results on a lexical search is a non-trivial task that requires the user to have an in-depth understanding of the code base. This comprehensive knowledge can be a lot to ask of one individual when working on a large software system. After a seed is formed, the fragility of a lexical search limits its effectiveness, as a lexical search simply searches for duplicates of its seed. Lexical analysis is not sufficient for finding duplicates in a programming text since “duplicate” often means a statement or method, or group of statements, with the same semantics. A group of statements under investigation as a possible duplicate could be non-contiguous, reordered, or intertwined with other groups in other locations[12]. In these cases, lexical search will fail to identify the duplicate. Lexical searches can also miss potential refactoring candidates in the following way. If one searches for all methods called “display” that have to do with outputting the object to the screen, several methods may be reported as expected, but there could also be a method that is semantically equivalent to the “display” methods that is called “putOnScreen”; this method will not be returned by lexical search, unless the user thinks to give that search seed also.

Exploratory tools can provide considerable help in developing an understanding of how a program works and allow a programmer to navigate more intelligently around code. The major disadvantage of these tools is that they require a lot of time to identify an aspect due to the required interaction with the user. The exploration is often incremental. The task of understanding the information being returned from the tool is left to the user along with intelligent use of that information. Also, a seed is required. This approach puts a heavy burden on the user.

In this paper, we present the design, implementation, and evaluation of an intelligent aspect mining analysis, which has the following properties:

- The analysis automatically identifies desirable candidates for refactoring into AOP.
- The approach is exhaustive, in that the analysis is performed on the entire code base.
- No input is required from the user to begin the analysis.
- No predefined queries are needed to start the analysis.
- A larger set of desirable refactoring candidates (including code matches that are reordered or non-contiguous) than with lexical analysis and other non-interactive analyses is identified.
- Space requirements are similar to internal program representations used for code optimization.
- Analysis time is reasonable in practice for large software systems, given that the analysis is offline, not within an interactive tool.

Our approach to automatic aspect mining exploits the program dependence graph and abstract syntax tree representations of a program. We have implemented the automatic mining technique within the Eclipse[5] environment. Using this system, an empirical evaluation has been performed on two large software systems to measure the cost and effectiveness of this approach to automatic aspect mining. As section IV demonstrates, the results are quite promising.

Section II presents our automatic aspect mining technique. Section IV describes the implementation and experimental evaluation. Section VI discusses conclusions and future directions for this research.

II. Automated Mining Technique

Our automatic aspect mining method consists of identifying initial refactoring candidates using a control-based comparison, followed by filtering based on data dependence information. The initial identification phase builds upon two existing algorithms that use the Program Dependence Graph (PDG) [6] to detect code clones[13], [12]. 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 property of a PDG that - only necessary orderings due to dependences are represented.

Komondoor and Horwitz[12] developed a tool that is able to identify duplicated code (i.e., clones) and display the clones to the programmer, who can then extract them into a separate new procedure and replace the duplicates with calls to the new procedure. Their algorithm first partitions the program’s statements (i.e., nodes in the PDG) into equivalence classes of matching nodes based on syntactic structure, ignoring variable names and literal values. For every pair of matching nodes, backward slicing is performed from each node in the pair, until predecessor nodes of the two slices do not match. A backward program slice on variable v at point p in a program is the set

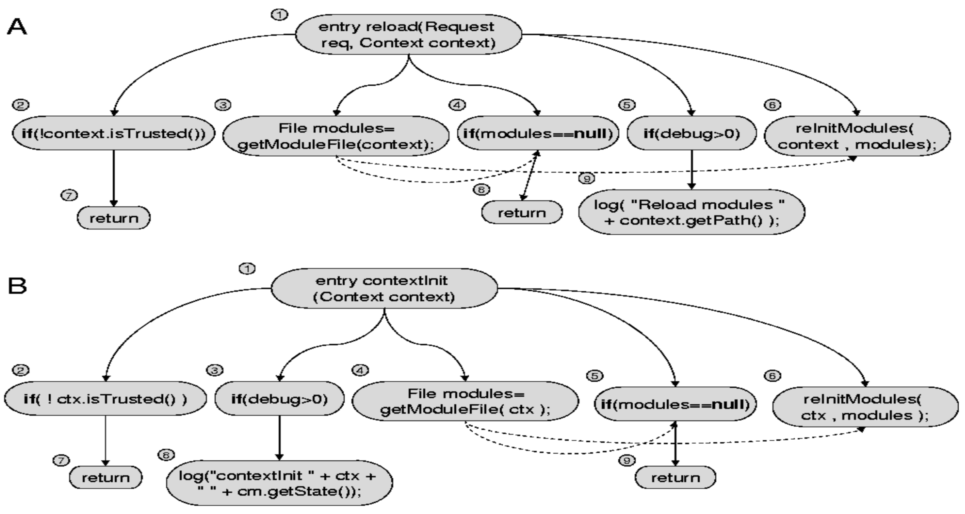

A
1
entry reload(Request req, Context context)
2
if(!context.isTrusted())
3
File modules= getModuleFile(context);
4
if(modules==null)
5
if(debug>0)
6
reInitModules(context , modules);
7
return
8
return
9
log("Reload modules " + context.getPath());
B
1
entry contextInit (Context context)
2
if(! ctx.isTrusted())
3
if(debug>0)
4
File modules= getModuleFile(ctx);
5
if(modules==null)
6
reInitModules(ctx , modules);
7
return
8
log("contextInit " + ctx + " " + cm.getState());
9
return

Fig. 1. PDGs of two code clones

of statements that directly or indirectly may affect the value of *v* at point *p*. Forward slicing is used to add matching nodes to the clones when matching predicate nodes are encountered to include matching control dependence successors of loops and conditionals. Thus, a pair of clones consists of two isomorphic subgraphs of the PDGs of two nodes in one of the equivalence classes. Subsequent steps of the algorithm remove clones subsumed by another clone, and combine pairs of clones into larger groups. Since this technique is capable of finding arbitrary clones that can be replaced by function calls automatically, it is general enough to be applied directly for aspect mining; however, the tool is admittedly slow, and the statement-level comparison is too course-grained for our aspect mining goals.

Krinke [13] developed a method for detecting similar, not necessarily identical, subgraphs of a specialized PDG representation of a program, with the motivation that often duplicated code is not identical, but has been adapted slightly to fit its context. Krinke's method is based on a specialization of a PDG in which there are nodes for expressions, variables and procedures as well as statements. Immediate control dependence edges exist between components of an expression to indicate the order of evaluation, while value dependence edges act as data dependence edges between expression components, and reference dependence edges represent when a computed value is stored into a variable. Using this more fine-grained PDG for two different procedures, and a pair of vertices v_i and v_j, one from each procedure's fine-grained PDG, Krinke's algorithm computes the maximal similar subgraphs of the two PDGs induced by k-limited paths starting at v_i and v_j. All possible matchings are considered at once, before adding the next set of reached vertices. While the technique achieves high precision (i.e., all reported duplicates are indeed duplicates), programs with large duplicated code cause exploding running times, and many duplicates are reported again due to their choice of starting nodes.

As a result of the high precision of these two clone detection approaches and their ability to identify clones that include non-contiguous parts and clones matching instances that have reordered parts, we have focused on adapting these techniques to obtain the benefits of each of these approaches for automatic aspect mining.

Our current analysis is based on AspectJ, arguably the most used AOP language available (just based on the number of books on AspectJ)[2]. While AspectJ 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[2].

In AspectJ, there are several types of *advice*, such as: “before”, “after”, and “around”. This advice (code) can be executed at a specified join-point. *Join-points* 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. Our current tool targets “before” advice that executes before a method in a specified set of methods is run. This type of advice is one of the most popular forms of advice.

Our algorithm for automatic mining of refactoring candidates for “before” advice precedes in four phases. The next subsections describe each of these phases in depth.

1) *Construct* source-level PDGs for all methods
2) *Identify* a set of refactoring candidates (control-based)
3) *Filter* undesirable refactoring candidates (data-based)
4) *Coalesce* related sets of candidates into classes

A. Constructing Source-level PDGs

Because our goal is to report candidates to the user for refactoring, our automatic aspect mining is performed on a source-level PDG, as opposed to a PDG typically built at the intermediate representation (IR) level. To reuse existing infrastructure for PDG construction, we construct a source-level PDG by first building a PDG at the IR level, where

```
public void contextInit( Context ctx )
throws TomcatException{
        // like a reload, the modules will
        // be removed and added back
        if( ! ctx.isTrusted() ) return;

        if(debug>0) log("contextInit " + ctx
                + " " + cm.getState());

        File modules=getModuleFile( ctx );
        if( modules==null ) return;

        reInitModules( ctx, modules );
}

public void reload( Request req, Context context)
throws TomcatException {
        if( ! context.isTrusted() ) return;
        File modules=getModuleFile( context );
        if( modules==null ) return;

        if( debug > 0 )
          log( "Reload modules " +
                context.getPath());

        reInitModules( context , modules);
}
```

Fig. 2. Example Code

control and data dependence information is used to construct the PDG according to traditional PDG construction[6]. To transform an IR-level PDG into a source-level PDG, nodes that correspond to the same source line number are collapsed into a single node. For each edge e from or to a node in the IR-level PDG which is subsumed, edge e is included in the source-level PDG appropriately from or to the subsuming node, avoiding duplicate edges being added to the source-level PDG.

B. Identifying an Initial Candidate Set

The second phase of our algorithm, which performs the identification of an initial refactoring candidate set takes a control-based approach, and can be viewed as a special case of automatic clone detection. General clone detection techniques work towards identifying clones that could be located at any point in a program, whereas "before" advice would be ultimately located at the beginning of a method. While the complexity of our problem will increase as we expand our research to capture "after" and "around" advice, the number of points in a program that we must consider are still substantially fewer than the general case. For instance, Krinke's algorithm[13] starts looking for clones at every predicate node. It is interesting to note that starting at predicates can result in not finding some clones. Komondoor and Horwitz[12] start at every statement in the program, making the approach non-scalable to large programs. In contrast, our algorithm starts only at the entry of each method.

Since the general problem of determining whether two subgraphs are isomorphic is NP-complete[13], the refactoring

```
Given: Program Dependence Graph A and B
Output: List matchingA; List matchingB;
Queue worksetA; Queue worksetB;
Node a = A.getEntryNode(); Node b=B.getEntryNode();
worksetA.add(a); worksetB.add(b);
while(worksetA.isEmpty()==false)
    Node currentA = worksetA.getAndExtractFront();
    List outGoingA = currentA.outControlEdges();
    Node currentB = worksetB.getAndExtractFront();
    while(outGoingA.iterator.hasNext())
        Node childA = outGoingA.iterator.next();
        List outGoingB=currentB.outControlEdges();
        while(outGoingB.iterator.hasNext())
            Node childB=outGoingB.iterator.next();
            if(childA.ASTMatches(childB))
                matchingA.add(childA);
                matchingB.add(childB);
                worksetA.add(childA);
                worksetB.add(childB);
return matchingA and matchingB;
```

Fig. 3. Candidate Identification Algorithm

candidate identification algorithm needs to make tradeoffs between accuracy in detecting clones and runtime performance. Komondoor and Horwitz[12] sacrificed accuracy to improve performance by using a very lightweight method for comparing two statements (i.e., PDG nodes) when placing them into the initial equivalence classes. In particular, they used a syntactic comparison, where variable and literal values were substituted with generic place-holders. Krinke's approach to reducing runtime requirements was to k-limit the allowable paths being investigated because the fine-grained PDG is so much larger than a traditional PDG.

Our algorithm for identification of the initial refactoring candidate set is similar to Komondoor and Horwitz in that it is also based on a traditional PDG representation of the program; however, no slicing is performed. Comparison of individual statement's Abstract Syntax Tree (AST) representations is utilized to improve accuracy, similar to Krinke's motivation for the fine-grained PDG. However, a separate program AST and a traditional PDG are used in place of the fine-grained PDG, and comparison is performed following control dependence edges only. Thus, our algorithm seeks to take advantage of the accuracy of Krinke's approach, but reduce the overhead during comparison and later coalescing.

Given two PDGs A and B representing two different methods A and B, the algorithm begins at the entry node of each PDG and performs a breadth first traversal of the control dependence subgraph of each PDG in lockstep, performing heavy pruning as it proceeds. The control dependence subgraph induces a tree structure on the PDG, with cycles for loops, which are handled by marking nodes during the traversal. The children of the entry node of PDG A are compared with each of the children of the entry node of PDG B. Any children that match, according to an AST comparison for the statement represented by the PDG node, are placed in a worklist to be expanded later, and in the set of matching nodes. After the entry nodes have been processed, a node is taken from the front of the worklist, and processed similarly. At any given state, where two nodes are being expanded in lockstep, $n * m$ new states can be created, where n is the number of children of the node being expanded in A, and m is the number of children of the node being expanded in B. The number of new states is reduced by pruning the matching of children that do not match. Since statement-level AST matching is fairly rigorous, many states are in fact pruned from the graph. The complexity of the identification process is also reduced because it starts only at the entry nodes of each PDG, not at many places in each method.

Figure 1 depicts two PDGs, labeled A and B, which are code clones of each other. Following the identification algorithm, line 5 sets `a` = A1 and `b` = B1. Then, `a` and `b` are added to the worklists for A and B. After entering the outer while loop, `currentA` and `currentB` are set to A1 and B1 respectively, after A1 and B1 are extracted from each worklist. The while loop on line 11 iterates through all of the children of A1, namely the set = {A2, A3, A4, A5, A6}. The while loop on line 14 iterates through the children of B1, namely the set = {B2, B3, B4, B5, B6}. On the first iteration of each while loop, A2 is compared to B2 at line 16. Since they match, these two nodes are added to the front of their corresponding worklists (to be expanded later). In the next iteration of the inner loop, A2 is compared to B3. Since there is no match, these nodes are not added to the worklist. This effectively terminates that possible search branch, and the state (A2,B3) will not be considered again. After this, A2 is compared to B4, B5, and B6 in that order, with no matches, and no additions to the worklists. The second iteration of the loop on line 11 starts. A3 is compared to B2 on the first iteration of the inner loop (line 16). A3 does not match B2, so nothing is added to the worklist. On the following iteration of the inner loop A3, is compared to B3, with no match, and then to B4, which it matches, so A3 and B4 are added to the end of their appropriate worklists. Assume that every iteration of the loop on line 11 is now complete for the first iteration of the loop on line 7. On the second iteration of the line 7 loop, `currentA` and `currentB` are removed from the front of the worklists. Since A2 and B2 were the first nodes to match, they are removed first. These nodes' children are then compared on the first iteration at line 16. A7 and B7 are compared, which match, and are added to the end of the worklists. The algorithm continues until there are no nodes left in A's worklist.

PDG nodes (i.e., statements) are compared for equality many times during the candidate identification algorithm, thus the comparison must be very fast. Therefore, the ASTs are simply compared on a structural level (except for literals and variables). This comparison is done by walking the ASTs of the two statements in lockstep. If any node is different from the corresponding node on the other statement's tree, then the trees are considered different. Literals of the same type (i.e., all boolean literals) are considered equivalent. Variables are equivalent if they are the same type, if one variable's type is a subtype of the other variable's type, or if they share a parent type besides Object.

This approach to AST comparison of statements has the following advantages: (1) Many of the negatives of a lexical comparison are avoided. Specifically, it will not fail because of whitespace, different variable names, and different literal values. It will still fail for semantically equivalent statements if they are reordered (e.g., i = 3+a not considered equal to i = b+3 where a and b are of the same type). (2) Two variables are considered to be equivalent if they are of the same hierarchy tree (the tree must be lower than the Object tree). This provides a stronger correlation between the matching of a statement and semantic equivalence.

C. Filtering Undesirable Candidates

The *Identify* phase may report potential refactoring candidates that are not desirable. Based on our experimental studies, we have determined several fast and inexpensive filters that can be performed on the initial refactoring candidate set to reduce the number of undesirable candidates reported to the user. The first filter is a data dependence based filter, which we will call *similar-data-dependence*. Given two PDG subgraphs that represent code clones, the *similar-data-dependence* filter determines whether the underlying data dependences are the same for each clone. Thus, the data dependence information is used (like the general clone detection methods), but as a filter after initial identification of clones. This provides a time savings overall.

Our second filter is the *outside-data-dependence* filter. Data dependence edges can also be used to eliminate candidates that have data dependences either to or from a node that is not part of the clone. AspectJ does not allow variables that are defined in "before" advice to be referenced in that method, so this type of candidate is not useful. We also perform a set of simple filters that eliminate trivial cases of undesirable refactoring candidates.

D. Coalescing Candidate Sets

The output from the *Identify* and *Filter* phases of our algorithm is a set of candidate pairs, where each member of a pair is from a different method. Since a method to method comparison of PDGs was performed in the *Identify* phase, it is possible that similar or even identical candidates have been reported in other method pairs. The *Coalesce* phase identifies these similar reported candidates and coalesces the pairs into sets of similar candidates. The final reported sets are more appropriate to identify scattered, duplicate code segments that would occur in more than two methods, often the case for aspects.

The coalescing is performed by comparing each reported candidate pair with one another using a lightweight (PDG) node by node comparison. The set of candidate pairs becomes a set of refactoring candidate classes as coalescing is performed. If a node n in the first instance of a given candidate class is found to match a node m in the first instance of another candidate class (according to the AST comparison), then n and m are marked "equal". If all nodes are marked "equal" after iterating through all of the first instance's nodes of each candidate class being compared, then the candidates of the two candidate classes are coalesced into a single refactoring candidate class. It might appear that dependences should be incorporated into this comparison, but in practice, this lightweight comparison has been effective, without the added expense of dependence comparison.

E. Analysis of Space and Time Costs

Time. Let m be the number of methods in the program being analyzed. The *Construct PDG* phase is performed for each method. PDG construction takes time O($n * e$) where n and e are the number of nodes and edges in a method's source-level CFG, respectively[6]. Thus, the PDG construction phase takes $O(n * e * m)$ time where n and e are averages for the PDGs in the program. The average number of edges and nodes in a PDG was very low for both of the programs that were investigated (see figure 6).

The *Identify* phase will perform m^2 PDG comparisons, where each PDG comparison could, at worst, take time proportional to the size of the number of nodes, n, in a PDG. In the worst case, a clone is identified between every method and each other method, creating m^2 clone objects. Each comparison requires at most $O(n)$ time since each node of the first PDG in a pair is compared with at most all of the siblings of its matching parent in the other PDG.

The *Filter* phase performs filtering on each candidate pair, and takes worst case $O(e)$ time for each filter as it is applied on each candidate, which has at most e edges. Thus, the filter phase takes $O(m^2 * e)$ time. In the *Coalesce* phase, there are at most m^2 candidate pairs to examine for coalescing. Coalescing takes $O(n)$ time, so this phase takes $O(m^2 * n)$ time. In this analysis, one AST comparison takes $O(a)$ time where a is the number of nodes in the AST of a single statement. Since the AST comparison involves only two statements at a time, a is a small constant.

The worst case time complexity is very unlikely in practice, given the experimental data we have collected and the situations that would cause the worst case. In practice, however, the entire program is usually not made up of many methods that all contain clones of each other. Programs should not have a large proportion of duplicate code. If so, additional time to analyze it and refactor would be worthwhile.

Space. The space requirements for our automated aspect mining consist of the PDG and AST for each method. The space for the source-level PDG is $O(N)$ where N is the number of lines in the source program. Similarly, space for the AST is $O(N)$. These are common data structures built and used during compilation for analysis and optimization. The only additional space is used for storing the refactoring sets themselves.

III. Implementation within Ophir

We have implemented our automated aspect mining analysis within Ophir, a framework that we have developed to support automatic mining analyses and manual or automatic refactoring of aspects. Ophir was developed as an Eclipse plug-in[5], and enables automatic aspect mining as follows. First, a user must choose a project to analyze by selecting the project in Eclipse's "Package Explorer" view. 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". 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.

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

Eclipse allowed us, because of its existing plug-ins and its extensible architecture, to easily build our automated mining analysis and the Ophir framework. We took advantage of the following Eclipse features: extensions points, plug-in independence, existing plug-ins, core plug-ins, GUI (Content Provider, Menu Provider, etc), and the refactoring environment. The FLEX[3] open research compiler infrastructure was used for constructing a source-level PDG. Since the PDG construction in FLEX is performed on byte-code, we used line number information contained in the byte-code to map to the source code level and create the source-level PDG.

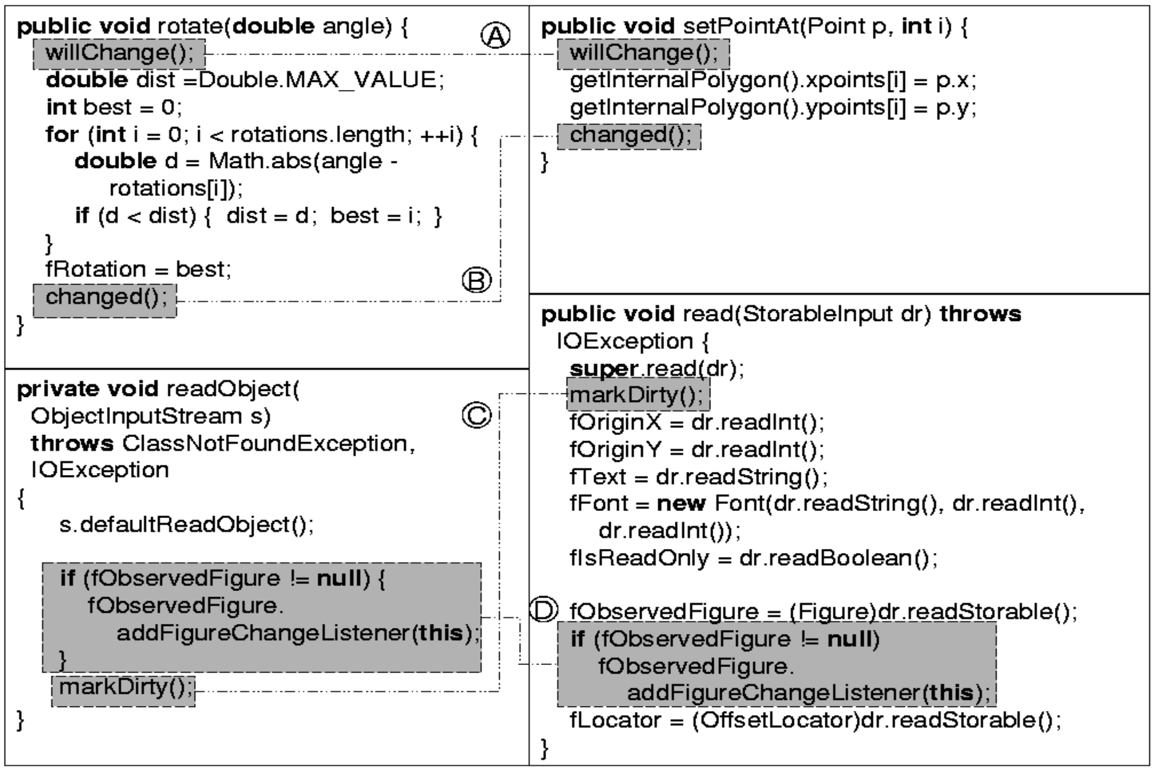
IV. Experimental Evaluation

A. Methodology

Characteristics	JHotDraw	Tomcat 3.2
No. of Files	195	430
No. of Methods	1,478	7,704
No. of Source Lines	12,307	38,495

Fig. 4. Benchmark Characteristics

For experimental evaluation, we executed our automatic aspect mining tool, on two reasonably-sized, realistic applications, JHotDraw[7] (ver. 5.3) and Tomcat[1] (ver. 3.2), which have been used as examples in many AOP projects. JHotDraw is a GUI-driven, technical drawing tool, similar to any simple painting program. Tomcat is a servlet container. To give some context to our experiments, Figure 4 shows the number of files, methods, and source lines in these applications. JHotDraw is smaller than Tomcat, and represents an example of a very clean implementation because it was built as a "design exercise". Tomcat is a larger, production-quality open-source project, and



```
public void rotate(double angle) {
  willChange();
   double dist =Double.MAX_VALUE;
   int best = 0;
   for (int i = 0; i < rotations.length; ++i) {
      double d = Math.abs(angle -
          rotations[i]);
      if (d < dist) {  dist = d;  best = i;  }
   }
   fRotation = best;
  changed();
}
```

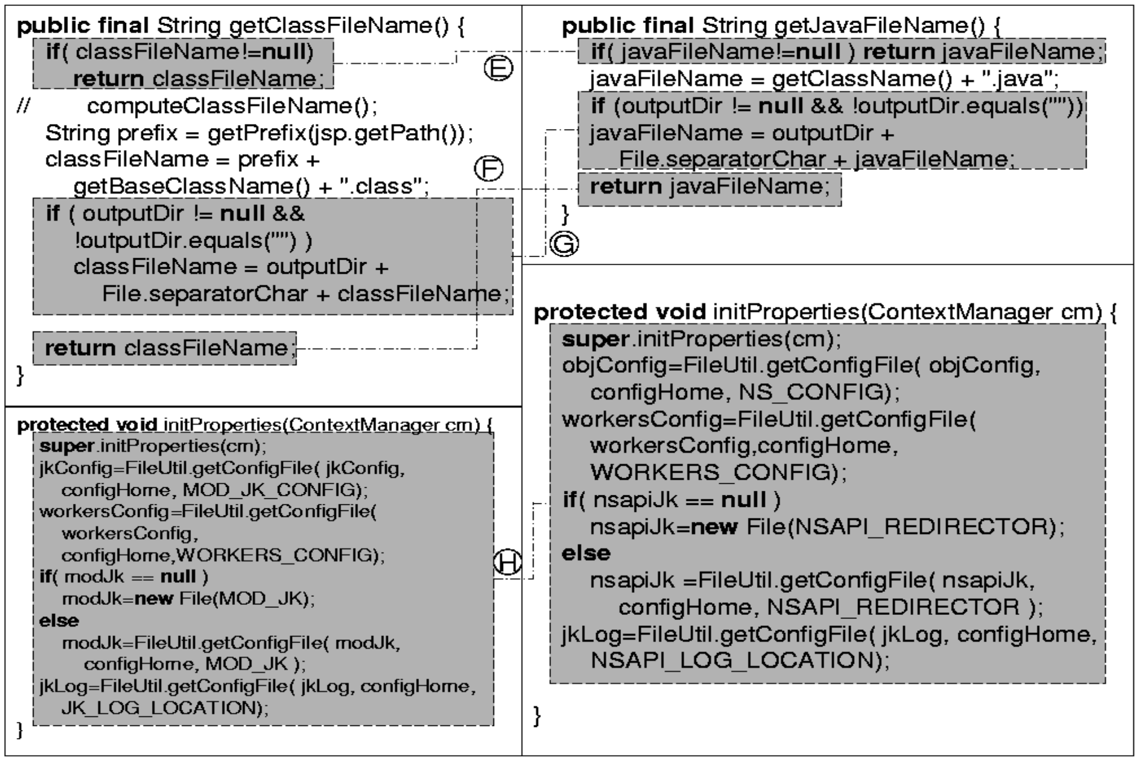
```
public void setPointAt(Point p, int i) {
  willChange();
  getInternalPolygon().xpoints[i] = p.x;
  getInternalPolygon().ypoints[i] = p.y;
  changed();
}
```

```
private void readObject(
 ObjectInputStream s)
 throws ClassNotFoundException,
 IOException
{
    s.defaultReadObject();

  if (fObservedFigure != null) {
     fObservedFigure.
        addFigureChangeListener(this);
  }
   markDirty();
}
```

```
public void read(StorableInput dr) throws
  IOException {
   super.read(dr);
   markDirty();
   fOriginX = dr.readInt();
   fOriginY = dr.readInt();
   fText = dr.readString();
   fFont = new Font(dr.readString(), dr.readInt(),
      dr.readInt());
   fIsReadOnly = dr.readBoolean();

   fObservedFigure = (Figure)dr.readStorable();
   if (fObservedFigure != null)
      fObservedFigure.
         addFigureChangeListener(this);
   fLocator = (OffsetLocator)dr.readStorable();
}
```



```
public final String getClassFileName() {
  if( classFileName!=null)
     return classFileName;
//       computeClassFileName();
  String prefix = getPrefix(jsp.getPath());
  classFileName = prefix +
     getBaseClassName() + ".class";
  if ( outputDir != null &&
     !outputDir.equals("") )
     classFileName = outputDir +
        File.separatorChar + classFileName;

  return classFileName;
}
```

```
  public final String getJavaFileName() {
    if( javaFileName!=null ) return javaFileName;
    javaFileName = getClassName() + ".java";
    if (outputDir != null && !outputDir.equals(""))
    javaFileName = outputDir +
       File.separatorChar + javaFileName;
    return javaFileName;
  }
```

```
protected void initProperties(ContextManager cm) {
  super.initProperties(cm);
  jkConfig=FileUtil.getConfigFile( jkConfig,
     configHome, MOD_JK_CONFIG);
  workersConfig=FileUtil.getConfigFile(
     workersConfig,
     configHome,WORKERS_CONFIG);
  if( modJk == null )
     modJk=new File(MOD_JK);
  else
     modJk=FileUtil.getConfigFile( modJk,
        configHome, MOD_JK );
  jkLog=FileUtil.getConfigFile( jkLog, configHome,
     JK_LOG_LOCATION);
}
```

```
protected void initProperties(ContextManager cm) {
  super.initProperties(cm);
  objConfig=FileUtil.getConfigFile( objConfig,
     configHome, NS_CONFIG);
  workersConfig=FileUtil.getConfigFile(
     workersConfig,configHome,
     WORKERS_CONFIG);
  if( nsapiJk == null )
     nsapiJk=new File(NSAPI_REDIRECTOR);
  else
     nsapiJk =FileUtil.getConfigFile( nsapiJk,
        configHome, NSAPI_REDIRECTOR );
  jkLog=FileUtil.getConfigFile( jkLog, configHome,
     NSAPI_LOG_LOCATION);

}
```

Fig. 7. Examples of Desirable Candidates

Candidate Characteristics	Mined Candidates			
	JHotDraw		Tomcat 3.2	
	count	%	count	%
Strong Candidate	54	56.25	131	54.58
Duplicate Method	20	20.83	38	15.83
Removed bySimple Filter	14	14.58	53	22.08
Analysis Required for Removal	8	8.33	18	7.50
Total	**96**		**240**	
% Useful Candidates*	90.24		90.37	

*The percentage is based on (Total - Removed by Simple Filter), as Removed by Simple Filter is a product of our prototype.

Fig. 5. Characteristics of Mined Candidates

PDG Space Characteristics	JHotDraw	Tomcat 3.2
Total Nodes	5,808	19,959
Ave. Nodes/PDG	5.27	6.77
Ave. Edges/PDG	6.83	10.24
Ave. Edges/Node	1.30	1.51
Phase	**Time in seconds**	
Construct	17	45
Identify	2,158	16,329
Filter	0	9
Coalesce	248	8,524
Total	**2,451**	**25,124**

Fig. 6. Analysis of Space and Time Costs

so its code, while well-written, is a better example of real world code.

For each application, Ophir returned a set of classes of refactoring candidates. Each class of refactoring candidates was manually examined in order to evaluate the effectiveness of the mining tool, and categorize the classes of mined refactoring candidates. Figure 5 presents the results from the manual inspection and categorization of the mined sets.

Each class of refactoring candidates was put into one of the categories: *Strong Candidate, Duplicate Method, Removed by Simple Filter*, or *Analysis Required for Removal*. If the class represented code that should be refactored into AspectJ, then it was categorized as a "Strong Candidate". If the code was essentially a whole duplicate method, it was put in the "Duplicate Method" category. The "duplicate method" candidates are easier to refactor than other candidates.

Sometimes Ophir identified classes of refactoring candidates that were not interesting for AOP. If a candidate was either a candidate that could be eliminated with a simple filter or a current implementation limitation, it was put into "Removed by Simple Filter". A simple filter is, for example, a post-pass that removes all classes that include empty clones. Another example of a simple filter is to remove all clones with only one node in which that node was an assignment to a local primitive variable. If a candidate class was a clone, but not appropriate for AOP and not trivial to remove without analysis, it was categorized as "Analysis Required for Removal".

B. *Identifying Desirable Candidates*

This subsection describes the findings from evaluating the effectiveness of our approach to automatic mining. Almost all of the candidates that Ophir mined were code that should be refactored into AspectJ (excellent accuracy). For JHotDraw, Ophir returned a total of 96 candidates, of which 74 were good candidates (categorized as either "Strong Candidate" or "Duplicate Method"), and 14 were only a product of our current prototype implementation. Disregarding the candidates caused by the current prototype's lack of the simple filters, our accuracy reaches 90.24% for JHotDraw. The percentage of mined candidates in Tomcat that are desirable is very similar. Since Ophir is the first automatic aspect mining tool requiring no seed, there is no other tool for comparison. However, the candidates that Ophir mines are overwhelmingly useful (over 90% with the addition of some simple filters in both cases).

Because of the size of the code (approximately 12,000 and 38,000 source lines, respectively), a human is unable to search manually for clones, and so it is difficult to determine which or how many candidates Ophir missed. Instead, we focused our manual inspection on candidates that Ophir identified that other existing tools could not find. Many candidates with this property were identified. In the rest of this section, we highlight some of these cases.

Figure 2 shows two methods that Ophir reported from Tomcat. There are several characteristics of these methods (which are essentially code clones of each other) that make them difficult to detect lexically. The variable names on line 5 and the corresponding line 18 are different. This shows that using variable names as a seed for lexically searching is undesirable, as it will miss some matches that need to be found. Lines 10 and 11 are identical to 19 and 20 semantically, but they are in a different contexts in each method (with different statements surrounding them). Lexically, these methods look fairly different, but according to the PDG, they are equivalent. It is easy to see that, starting at node A-1 and B-1 in figure 1 that nodes A-3 and B-4 are found to be matches because, in the candidate identification algorithm, every child of A-1 is compared with every child of B-1. The PDGs lack of strict ordering enables it to find this clone.

In addition to demonstrating several of the kinds of candidates that are often found by Ophir, this particular example shows candidates that are desirable for AOP. First, on lines 5 and 18, a good example of a policy is shown. This policy can be expressed informally as "when a context is not trusted, do not perform a contextInit or a reload with it". Policies, like this, are good candidates to be refactored as aspects. In this example, debugging code is also apparent in lines 7, 8, 22, 23, and 24. Debugging is a classic example of code that could be refactored into an aspect. Lines 10, 11, 19, and 20 are the core of these two methods. If one were to refactor these two methods, they should be merged into one method, with these lines as the core. The method calls at the end of each method (lines 13 and 26) are calls to "cleanup" methods. It is easier to understand the concept of a "cleanup" method after studying figure 7 and noticing the pair marked “B”. A “cleanup” method is essentially code used to clean up or propagate the changes that a method just caused in the system. When a figure is moved in a figure editor, a “cleanup” method is often called to update the display.

One might ask how the candidates in figure 2 found by Ophir could be found by other tools. Starting with no knowledge of the code base, they cannot be found. These are two methods hidden within thousands. One could search for all uses of the type Context (of course, how did one arrive at this decision, unless they wished to do this for every type), but this returns a huge superset of these two methods, and in the process of manually examining them all, one would probably not recognize these as clones (because of sheer volume). One could also, after exploring the code, realize that logging is scattered in many places, and search for the log method. This would produce good results, but logging is probably one of the most obvious aspects. One still would not recognize these two methods as clones, because in refactoring all of the “log” calls, one would have to look at many other methods, again passing over these two without recognizing that they are clones. One could also search for all the calls to “reInitModules”, which should probably be refactored to an aspect. However, how would one decide to do this? You would certainly not want to search for every method’s calls in a program, as that is too time consuming. You would probably have to scan enough code to run across several examples of reInitModules being used as a “cleanup” method (at the end of the method). These calls would have to be lexically at the end of each method for a human to recognize them quickly as a cleanup method. This task could be extremely time consuming, and humans are error prone. This example is meant to show how frustrating it could be for a programmer to refactor a program into AOP. The programmer is forced to read or interactively browse and understand a large part of the source code before these aspects become apparent.

Figure 7 shows several examples of desirable refactoring candidates mined by Ophir. Some of these examples have already been discussed. Candidates A and B represent more classic examples within JHotDraw of code that should be an aspect. The method calls are used to let the system know when an object that is being displayed is changed. To the trained human, A and B are obviously good candidates. However, a human still might have to go through thousands of files before stumbling across these candidates. Candidate C represents a cross-cutting concern of marking objects dirty after reading from them. Candidate D represents the cross-cutting concern of channel listener management. Both of these candidates would be difficult for a human to catch, because they are at different places in different methods lexically.

C. *Costs of Automatic Mining*

A tool like Ophir is meant to be run infrequently. It could be run overnight about once a month in order to keep a code base nicely organized, readable, and easier for future maintenance. Even with our prototype implementation, Ophir ran in a reasonable amount of time. Figure 6 indicates the times for each phase and the total analysis time. Note that the total times include other processing time that is not included in the individual phases. JHotDraw took only 41 minutes to analyze, while Tomcat, which has a considerably larger PDG representation, took a little under 7 hours. Not surprisingly, most of the time spent on each program was in the *Identify* phase. However, this cost could, in the future, be trimmed dramatically, since each of the m^2 comparisons could be done in parallel. Also, our current prototype was considerably slowed down by the way we implemented the AST comparison, as we could not identify an efficient way to access the AST within Eclipse for a particular line number in a file.

The characteristics for the constructed PDGs are presented in figure 6. In practice, both of our programs had low averages for number of edges and nodes in each method’s PDG. This is important as in the analysis section, many of the worst case space and time requirements are based on the size of a PDG for a given method.

V. Related Work

A. *Aspect Mining Tools*

The Aspect Browser [8] was one of the first tools available. It was purely lexical, functioning much like the Unix utility “grep”. However, it included a nice graphical representation of its search results, making it easy to see how a concern was scattered. To improve the effectiveness of the tool, the developers suggested that one use “information transparency” in which the coder uses lexical means (like naming methods and variables) to relate pieces of code across class boundaries.

The Aspect Mining Tool (AMT)[9] is similar to the Aspect Browser, but it uses type information which helps in very specific cases. However, AMTex[19], an extension of the AMT, uses type information in a more interesting way, ranking all types based on how scattered they are throughout the code base. This ranking helps the programmer find scattered concerns.

The tools JQuery[18] and FEAT [16] are both exploration tools, which are fundamentally different than the previous tools. They help to guide the user through an exploration process in order to help the user make deductions about which code is part of a concern. FEAT also includes a representation of concerns, called the Concern Graph, which uses nodes in the AST (like a method or a variable) and relationships between these nodes (like x calls y) to represent a concern. Once one gets a starting node in the AST (from a lexical search), the graph can be incrementally grown via queries (e.g. who does x call). JQuery uses a functional query language to help speed up this incremental process. One can form functional queries on the “program database”, which includes facts like X calls Y and Z is a variable in O. These queries allow one to form different types of browsers within the same framework. The framework keeps a history of the incremental path that has been traveled while searching. Both of these tools require a lot of intuition from the user to be used for aspect mining.

B. Clone Detection Techniques

Several general methods for detecting clones exist, and can be categorized as metrics-based, token matching, or AST comparison. For example, the Covet [14] tool uses a comparison of function metrics to find code that is similar. Token matching (after a language dependent parse) is used by CCFinder [10] and JPlag [15] to detect similar patterns of tokens. CloneDr [4] uses a comparison of the AST of a program to find matching subtrees (clones).

VI. Conclusions and Future Work

We have shown how the PDG and the AST can be used to mine desirable candidates from large systems in a reasonable amount of time. Our tool is able to identify desirable refactoring candidates without requiring program knowledge or input from the user. Out of the candidates that were automatically identified in two large software systems, 90% or more were classified as desirable refactoring candidates. Since there exists no other automatic aspect mining tool for comparison and the code bases used as input were too large to search manually with accuracy, it is difficult to estimate recall (candidates found / existing candidates). We plan to improve the performance of the tool by exploiting parallelism in the second phase. While our tool can be run on code already in AOP by representing advice as methods, we are currently investigating other approaches to mining within AOP code.

References

[1] Tomcat homepage, http://jakarta.apache.org/tomcat/. (October, 18, 2003).

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

[3] Flex's homepage, <http://www.flex-compiler.csail.mit.edu/harpoon/>. 2003.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, 1997.

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

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its uses in optimization. In *ACM TOPLAS*, July 1987.

[7] E. Gamma and T. Eggenschwiler. JHotDraw homepage, http://www.jhotdraw.org/. (October 18, 2003).

[8] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Workshop on Multi-Dimensional Separation of Concerns*, 2000.

[9] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advances Separation of Concerns*, 2001.

[10] T. Kamiya, F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue. Maintenance support tools for Java Programs: CCFinder and JAAT. In *ICSE*, 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP*, 1997.

[12] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Static Analysis Symposium (SAS)*, July 2001.

[13] J. Krinke. Identifying similar code with program dependence graphs. In *Eight Working Conference On Reverse Engineering*, October 2001.

[14] J. Mayrand, C. Leblanc, and E. Merlo. Automatic detection of function clones in a software system using metrics. In *ICSM*, 1996.

[15] L. Prechelt, G. Malpohl, and M. Phillippsen. JPlage: Finding plagiarisms among a set of programs. In *Technical Report 2000-1*.

[16] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *ICSE*, 2002.

[17] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[18] K. D. Volder and D. Janzen. Navigating and querying code without getting lost. In *AOSD*, 2003.

[19] C. Zhang, G. Gao, and A. Jacobsen. Amtex homepage, <www.eecg.utoronto.ca/ czhang/amtex>. (October 18, 2003).