Automated Inference of Pointcuts in Aspect-Oriented Refactoring

Prasanth Anbalagan[1] Tao Xie[2]
Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
[1]panbala@ncsu.edu [2]xie@csc.ncsu.edu

Abstract

Software refactoring is the process of reorganizing the internal structure of code while preserving the external behavior. Aspect-Oriented Programming (AOP) provides new modularization of software systems by encapsulating crosscutting concerns. Based on these two techniques, aspect-oriented (AO) refactoring restructures crosscutting elements in code. AO refactoring includes two steps: aspect mining (identification of aspect candidates in code) and aspect refactoring (semantic-preserving transformation to migrate the aspect-candidate code to AO code). Aspect refactoring clusters similar join points together for the aspect candidates and encapsulates each cluster with an effective pointcut definition.

With the increase in size of the code and crosscutting concerns, it is tedious to manually identify aspects and their corresponding join points, cluster the join points, and infer pointcut expressions. Therefore, there is a need to automate the process of AO refactoring. This paper proposes an automated approach that identifies aspect candidates in code and infers pointcut expressions for these aspects. Our approach mines for aspect candidates, identifies the join points for the aspect candidates, clusters the join points, and infers an effective pointcut expression for each cluster of join points. The approach also provides an additional testing mechanism to ensure that the inferred pointcut expressions are of correct strength. The empirical results show that our approach helps achieve a significant reduction in the total number of pointcut expressions to be used in the refactored code.

1. Introduction

Aspect-oriented programming (AOP) [15] provides constructs for modularizing crosscutting concerns, i.e., those functionalities that are scattered among many modules in a system. AspectJ [16, 12, 19], an aspect-oriented programming language, is an extension to Java. AspectJ provides new constructs like pointcuts, join points, advice, and aspects to help modularize crosscutting concerns. *Join points* are well defined locations in the execution of a program. *Pointcuts* are those constructs that help identify these join points in code. A pointcut consists of *pointcut expressions* combined with logical operators. *Advice* contains the set of actions to be taken once the pointcut matches the join points. The encapsulation of join points, pointcuts, and advice is called as an *aspect*.

Software applications often contain many instances of crosscutting concerns. Refactoring such applications toward AOP (also known as *aspect-oriented refactoring* [17]) helps modularize these crosscutting functionalities. Aspect-oriented refactoring consists of identifying the crosscutting concerns in an application and representing them using appropriate AOP language mechanisms. Representing identified crosscutting concerns (also known as *aspect refactoring*) using AspectJ constructs like join points and pointcuts requires identifying the join points for each crosscutting concern and representing them with pointcut expressions. Pointcuts provide the usage of logical operators that combine the individual pointcut expressions. The pointcut expressions combined together form a pointcut that represents an aspect.

In real-world applications, the source code would span to thousands of lines and it would be tedious to manually identify aspects and their corresponding join points, cluster similar join points together, and infer pointcut expression for join-point clusters. With the increase in the size of the source code and concerns, if we represent the concerns by simply combining the pointcut expressions for join points using logical operators, the resulting pointcuts would be inefficient when being written, compiled, and executed, and would be difficult to maintain. In this paper, we make the following main contributions:

- We propose an automated approach for mining aspects, identifying their corresponding join points in source code, clustering the join points based on a common characteristic, and inferring an effective pointcut expression to represent each cluster of join points.
- We implement a tool to support our approach. The tool consists of four main components: the aspect mining module, join point identifier, clustering module, and inference module. The tool also provides an additional testing mechanism to ensure that the inferred pointcuts

are of correct strength, i.e., the pointcuts match only the intended set of join points.

- We evaluate our approach on six applications taken from a variety of sources like AspectJ benchmark suites[1] and SourceForge[2]. The empirical results show around 34% to 81% reduction in the number of pointcut expressions required to refactor crosscutting concerns to aspect-oriented code. The empirical results also show that our approach could suggest possible refactoring of method names (related to join points) for achieving further reduction.

The rest of the paper is organized as follows. Section 2 presents our illustrative example. Section 3 illustrates our approach. Section 4 describes the implementation of the approach. Section 5 provides the results of applying the approach on selected applications. Section 6 discusses related work, and Section 7 concludes.

2. Example

Our approach complements the existing aspect-mining tools and techniques [14] used for aspect-oriented refactoring [17], serving as a post-processor of the results produced by these aspect-mining techniques. The implementation of our approach is built around AspectJ [16, 12, 19], an AOP language. We next use an example to illustrate AspectJ, its various features, and aspect-oriented refactoring to AspectJ code with the support of our approach.

AspectJ [16, 12, 19] is a simple and practical aspect-oriented extension to Java. With just a few new constructs like join points, pointcuts, advice, and aspects, AspectJ provides support for modular implementation of a range of crosscutting concerns. Join points are well-defined locations within the primary code where a concern crosscuts the application. The join points in AspectJ include method or constructor calls, method or constructor execution, the initialization of a class or object, field reads, field writes, exception handler executions, class initializations, and object initializations.

Pointcuts are predicates that match events (join points) in the execution of a program, i.e., they specify where the crosscutting behavior applies. Pointcuts are modelled using expressions that identify the type, scope, or context of the events. AspectJ pointcuts provide the features of abstraction and composition, which include various designators, wildcards, and their combination with logical operators. Advice contains its own set of rules as to *when* it is to be invoked in relation to the join point that has been triggered. Advice specifies what the crosscutting behavior does. For example, *before* advice is executed before the join point is executed,

[1] http://www.sable.mcgill.ca/benchmarks/

[2] http://sourceforge.net/

```
aspect beforeDrawApplication {
 private pointcut invokeTool():
 (execution(public void DrawApplication.prompt*()) ||
            public void DrawApplication.*Selection(..))

 before(): invokeTool() {
      DrawingView.toolDone();
 }
}

aspect afterDrawApplication {
 private pointcut changeSelection():
 (execution(public void DrawApplication.*Selection(..));

 after(): changeSelection() {
      DrawingView.fireSelectionChanged();
 }
}

aspect newDrawApplication {
 private pointcut markText():
  (execution(public void DrawApplication.read*(..)) &&
   set(boolean DrawApplication.isTextDirty));

 after(): markText() {
      DrawingView.markTextDirty();
 }
}
```

Figure 1. Sample aspects for the `DrawApplication` class

and *after* advice is executed after the join point is executed. Aspects are modular units of crosscutting implementation. The encapsulation of join point, pointcut, and advice is provided by an aspect.

Figure 1 shows three sample aspects for a `DrawApplication` class, which is a class refactored from the class shown in Figure 2. For example, the first aspect `beforeDrawApplication` includes a piece of `before` advice, which is to be invoked *before* a method whose name is matched with `public void DrawApplication.prompt*()` or `public void DrawApplication.*Selection(..)` is executed, where ".." denotes any number of arguments.

However, in practice, a large amount of legacy code with substantial crosscutting concerns was not written in AOP languages and therefore cannot enjoy the benefits provided by AOP languages such as AspectJ. Then aspect-oriented (AO) refactoring [17] can be used to organize a crosscutting concern into aspects, i.e., moving all code responsible for implementing a particular crosscutting concern into an aspect. In AO refactoring, an important step is to identify what elements are to be refactored and what aspect solutions can replace them. At the aspect mechanism level, these elements to be refactored are the crosscutting concerns and the solutions that replace them are aspects. To automate the process of aspect-oriented refactoring, we need to identify the join points (the locations of the crosscutting concerns), and represent them using pointcut expressions. We next illustrate how our approach automatically supports aspect-oriented refactoring through the example `DrawApplication` class shown in Figure 2.

```
public class DrawApplication extends ... {
   public void promptNew() {
        DrawingView.toolDone();
        ...
   }
   public void promptOpen() {
        DrawingView.toolDone();
        ...
   }
   public void clearSelection() {
        DrawingView.toolDone();
        ...
        DrawingView.fireSelectionChanged();
   }
   public void toggleSelection(Figure figure) {
        DrawingView.toolDone();
        ...
        DrawingView.fireSelectionChanged();
   }
   public void read(StorableInput dr) {
        ...
        isTextDirty=true;
        DrawingView.markTextDirty();
        ...
   }
   public void readObject(ObjectInputStream s) {
        ...
        isTextDirty=false;
        DrawingView.markTextDirty();
        ...
   }
   ...
}
```

Figure 2. The `DrawApplication` class before being refactored

In this `DrawApplication` class, our approach can automatically identify three aspects[3](crosscutting concerns) being spread across multiple methods of the class. These three aspects are static method calls to `DrawingView.toolDone()`, `DrawingView.fireSelectionChanged()`, and `DrawingView.markTextDirty()`, respectively. In order to refactor these functionalities, we need to identify the locations in the code where these aspects occur, i.e., we need to capture all the join points that include the call sites of `toolDone()`, `fireSelectionChanged()`, and `markTextDirty()`. In practice, the aspects might occur at the beginning or end of a method or in the middle of a method with statements before or after the aspects. For example, the aspect related to `markTextDirty()` involves write access to a field `DrawApplication.isTextDirty` before `markTextDirty()`'s execution. Hence the write access to `DrawApplication.isTextDirty` is also captured as a join point for the aspect related to `markTextDirty()`. Our approach can automatically identify the captured join points for these three aspects as below:

```
Join points for DrawingView.toolDone(): before
 1.public void DrawApplication.promptNew()
 2.public void DrawApplication.promptOpen()
 3.public void DrawApplication.clearSelection()
 4.public void DrawApplication.toggleSelection(Figure)
```

[3]In this paper, we use the term of aspects in a broad sense to mean crosscutting concerns beyond the aspect constructs in AspectJ.

```
Join points for DrawingView.fireSelectionChanged():
after
 5.public void DrawApplication.clearSelection()
 6.public void DrawApplication.toggleSelection(Figure)
Join points for DrawingView.markTextDirty(): after
  set boolean DrawApplication.isTextDirty) in
 7.public void DrawApplication.read(StorableInputStream)
 8.public void DrawApplication.readObject(ObjectInputStream)
```

where *before* and *after* denote the points before and after executing or calling an affected method or setting a field (denoted as *set*), respectively.

To minimize unwanted effects, a straightforward refactoring would generate pointcuts that simply enumerate each of the affected methods, i.e., the pointcut expressions (each being the name and signature of the affected method) for each join point are combined with the `OR` logical operator "`||`". But aspects are usually spread across multiple classes and methods. With a large number of concerns, it is not efficient to combine the pointcut expressions for the join points of an individual concern with a logical operator.

Our approach automatically clusters join points based on common characteristics in the method names in these join points. For example, among the join points for the aspect related to `DrawingView.toolDone()`, our approach clusters join points 1 and 2 together because the method names there share the same prefix "`prompt`", and clusters join points 3 and 4 together because the method names there share the same postfix "`Selection`".

Then for each cluster of join points, our approach automatically infers an effective pointcut expression that could represent each set of join points. For example, from the preceding two clusters, our approach infers two pointcut expressions: `public void DrawApplication.prompt*()` and `public void DrawApplication.*Selection(..)`. Then we can use "`||`" to combine these two inferred pointcut expressions to form an effective pointcut (like the manually written one shown in Figure 1), rather than using "`||`" to combine four pointcut expressions derived directly from the method names and signatures in the four join points. In addition, our approach uses our pointcut testing framework [3] to verify and ensure that the inferred expression matches only the intended set of join points and not any unintended join points because of generalization.

3. Approach

The overview of our approach is shown in Figure 3. Our approach consists of four main components: the aspect mining module (Section 3.1), the join point identifier (Section 3.2), the clustering module (Section 3.3), and the inference module (Section 3.4). The aspect mining module identifies aspects (crosscutting concerns) and the join point identifier identifies the join points for each aspect. Along with the join points, the `before` and `after` code preced-

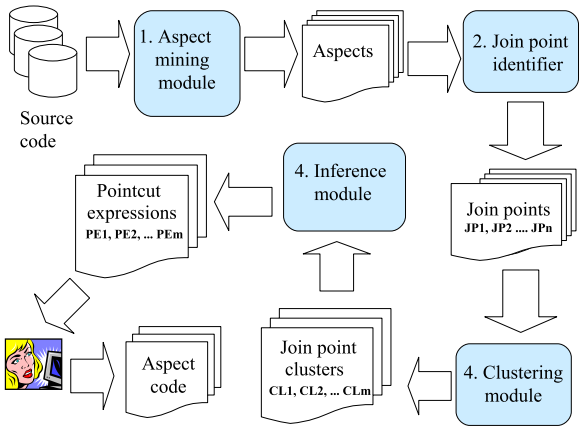

Figure 3. Overview of our approach

ing or following the join points are collected to form the `before` or `after` advice. The clustering module groups join points into clusters based on their common characteristics. The inference module operates on each of these join point clusters to infer a pointcut expression to represent the cluster of join points.

3.1. Aspect mining module

The aspect mining module uses existing aspect mining tools to identify aspects in given source code. Existing aspect mining tools have been developed for mining aspects as well as hidden concerns in general. These tools help identify tangled code. Repeated usage of a specific type (such as a method and variable) in interwoven code is a possible indication of tangled code. From source code, the tools extract aspects (crosscutting concerns), which are to be processed by the join point identifier.

3.2. Join point identifier

The join point identifier operates on the list of aspects output by the aspect mining module and identifies their corresponding locations in the source code. At the AspectJ mechanism level, locations are join points, which include a method or constructor call (a method or constructor of a class is called), method or constructor execution (an individual method or constructor is executed), the initialization of a class or object, field get (a field of an object, class or interface is read), field set (a field of an object, class or interface is set), exception handler execution (an exception handler is executed), and class initialization (static or dynamic).

To effectively represent an aspect (crosscutting concern) with a pointcut expression, we need to identify exactly the join points for that aspect, i.e., apart from identifying the location of the statement that has the aspect, we need to ana-

```
aspect beforeDrawApplication {
 private pointcut invokeTool() :
 (execution(public void DrawApplication.promptNew()) ||
  execution(public void DrawApplication.promptOpen() ||
  execution(public void DrawApplication.clearSelection()) ||
  execution(
     public void DrawApplication.toggleSelection(Figure));
 before(): invokeTool() {
     DrawingView.toolDone();
 }
}
```

Figure 4. `beforeDrawApplication` **Aspect**

```
aspect afterDrawApplication {
 private pointcut changeSelection() :
 (execution(public void DrawApplication.clearSelection()) ||
  execution(
     public void DrawApplication.toggleSelection(Figure)));
 after(): changeSelection() {
     DrawingView.fireSelectionChanged();
 }
}
```

Figure 5. `afterDrawApplication` **aspect**

```
aspect newDrawApplication {
 private pointcut markText() :
 (execution(
   public void DrawApplication.read(StorableInput)) ||
  execution(
    protected void DrawApplication.readObject(
                                ObjectInputStream))) &&
  set(boolean DrawApplication.isTextDirty));
 after(): markText() {
     DrawingView.markTextDirty();
 }
}
```

Figure 6. `newDrawApplication` **aspect**

lyze the statements before and after the aspect. The location of an aspect can be one of the three cases:

1. The crosscutting concern to be moved to the aspect is at the *beginning* of a method with no statements before the concern but has statements after the concern.
2. The crosscutting concern is at the *end* of a method with statements before the concern but no statements after the concern.
3. The crosscutting concern is in the *middle* of a method, with statements before and after the concern.

The crosscutting concern `DrawingView.toolDone()` shown in Figure 2 is an example for Case 1, where the concern is at the beginning of four methods. The pointcut for this type of concern would be the complete signature of each method with the designator as `execution`. The method call `DrawingView.toolDone()` can be placed in the `before` advice of the aspect that has the pointcut expression shown in Figure 4.

In the `DrawApplication` class (Figure 2), the method call to `DrawingView.fireSelectionChanged()` is found at the end of two methods. This example falls into Case 2, where the crosscutting concern is at the end of a method with no statements after the concern. In this case, the method to be refactored is placed in the `after` advice of the aspect, as shown in the aspect `afterDrawApplication` in Figure 5.

In the `DrawApplication` class (Figure 2), the method call to `DrawingView.markTextDirty()` is found in the middle of two methods. This example falls into Case 3, where the crosscutting concern is in the middle of a method, with statements before and after the concern. We need to include the statement before or after the concern in the pointcut expression in order to identify the location of the concern precisely.

In this case, the method to be refactored is placed in the `after` advice of the aspect, as shown in Figure 6. The pointcut definition includes the write access to the field `DrawApplication.isTextDirty` since this statement commonly precedes the concern. Our implementation identifies the module (class or method) to which the join point belongs and then collects the before and after code depending on the case under which the join points fall into. The before or after code is then analyzed to determine the type of join point they belong to (like execution or call to a method, and read or write access to a field). In our current implementation, the join points are formed based on static analysis of the code and we currently do not support forming join points for dynamic context such as `cflow`.

It is possible that the statement before or after the concern may be repeated more than once in the same method body and other occurrences of the statement are not associated with the concern. This type of join points cannot be easily captured by AspectJ's existing pointcut language, and it is not handled in our current implementation either. When this case occurs, aspect-oriented refactoring cannot be conducted on the locations of the join points unless developers refactor these locations to allow the join points to be captured with pointcuts.

3.3. Clustering module

The clustering module receives the aspect candidates and their join points as input. The core functionality of this component is to identify the common characteristic among the join points and group them into clusters. In order to perform clustering, the attribute identifier in the module processes the join points to identify their attributes. Then the clustering engine in the module performs clustering based on the attributes and outputs the join points along with their cluster number. Then the join points under each cluster are grouped together to form the input to the inference module.

In particular, we perform clustering based on attributes such as different naming parts of a join point for an aspect. For example, if a join point is that of a method execution, then the attributes would be naming parts of the method, i.e., its modifiers, return type, class name, method name, and arguments. By default, our approach automatically chooses the prime factors based on which the grouping of join points is performed. Our current implementation chooses the method-name field of a join point as the prime factor. The method-name field of a join point is split into different parts. Splitting the method-name field gives a higher probability of detecting a common pattern among the method-name fields of different join points. For example, consider the method names of the join points `public void clearSelection()` and `public void toggleSelection(Figure figure)` in Figure 2. The names are split into "`clear`", "`Selection`", and "`toggle`", "`Selection`". After the names are split, we find that there is a common pattern "`Selection`" and the method names are grouped into one cluster. The same procedure of splitting name fields is repeated for all join points. We form input to the clustering engine by including the split name fields and the individual join points.

In the case of join points for the aspect `DrawingView.toolDone()`, two clusters are produced because the methods `public void promptNew()` and `public void promptOpen()` have a common pattern "`prompt`" in their names, and the methods `public void clearSelection()` and `public void toggleSelection(Figure figure)` have a common pattern "`Selection`". The cluster assignment is shown in Figure 8, where each join point is assigned a cluster number. The clustering module further processes the cluster assignment to group the join points belonging to a cluster together. The clustered join points are then fed as input to the inference module.

3.4. Inference module

Figure 7 shows an overview of the inference module. The inference module generates pointcut expressions given a set of clustered join points. The designator identifier in the module identifies the type of designators (such as `execution` and `call`) for the join points. The inference module identifies the naming parts of the clustered join points, forms a regular expression for each set of naming parts, and finally outputs the pointcut expression by combining the individual expressions with the pointcut designator generated by the designator identifier. The inference module also provides an additional testing mechanism to verify the strength of the inferred pointcuts.

In the clustered join points given to the module as input, the naming parts of join points are separated by commas. The field separator in the module parses each item in the input and identifies the naming parts. The output from the field separator is fed as input to the string aligner. The string aligner infers expressions for each naming part and then combines them with the identified designator to form a complete pointcut expression.

The string aligner is based on a string alignment algorithm. Given a set of strings, the aligner performs alignment

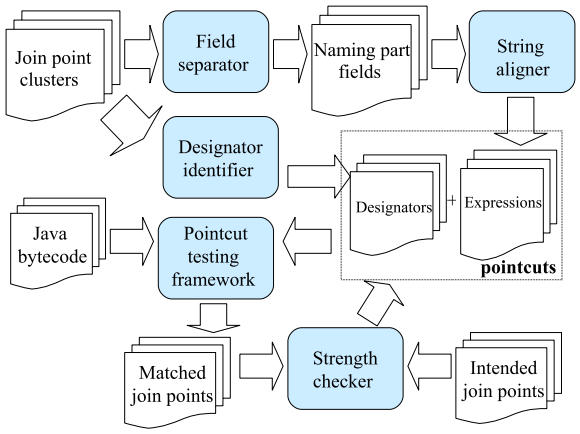

Figure 7. The Inference module

```
Sample result:
 Class : DrawApplication (Figure 2)
 Join points for aspect toolDone():
   public void promptNew()                    (cluster1)
   public void promptOpen()                   (cluster1)
   public void clearSelection()               (cluster2)
   public void toggleSelection(Figure)        (cluster2)
 Pointcut expression:
   execution(public void DrawApplication.prompt*())
   execution(public void DrawApplication.*Selection(..))
 Join points for fireSelectionChanged():
   public void clearSelection()               (cluster3)
   public void toggleSelection(Figure)        (cluster3)
 Pointcut expression:
   execution(public void DrawApplication.*Selection(..))
 Join points for markTextDirty():
   public void read(StorableInput)            (cluster 4)
   public void readObject(ObjectInputStream) (cluster 4)
 Pointcut expression:
   execution(public void DrawApplication.read*(..))
```

Figure 8. Sample clustering and inference results

of the strings with each other and infers an expression. If more than one expression can be inferred, an optimal one is selected. The optimality is determined as the expression that suits best for all strings in the cluster as well as having the maximal similarity measure with the strings based on a string similarity measure. The inferred expression is combined together with the designator corresponding to the join points.

Figure 8 shows the inference results for the example. Figure 1 shows the aspects from Figures 4, 5, and 6 with pointcuts replaced by the pointcut expressions inferred by our approach.

The inferred pointcut expression and the Java bytecode of the system under refactoring are analyzed using our pointcut testing framework [3]. The framework outputs the set of join points that are actually matched by the inferred pointcut expression. The set of actually matched join points (output by the framework) and the original set of intended join points (from which the pointcut expression is inferred) are fed as input to the strength checker, which compares the two sets of join points. If the sets are equal, then the inferred pointcut expression is of correct strength. If the sets are not equal (i.e., the actually matched join point set is a proper superset of the original intended join point set[4]), then there are two main options to address the incorrect strength. First, with the information and tool support provided by our approach, developers can easily refactor both the inferred pointcut expression and its intended join points to make them more specific (e.g., adding specific keywords to their names) so that the new pointcut expression matches only the intended join points. Second, the pointcut expression of incorrect strength can be appended with expressions like "`&& (!jp)`", where `jp` is an unintended but matched join point, to exclude `jp` from being matched by the new pointcut expression. Note that we can additionally infer an effective expression to match these unintended but matched join points to reduce the complexity of the appended expressions. But if the appended expressions induce a more complex pointcut expression, we can fall back to the straightforward way of simply combining intended join points (in the cluster for the pointcut expression) using the `OR` logical operator.

4. Implementation

We have implemented our approach for AspectJ and Java code using the Aspect Mining Tool (AMT) [13], Byte Code Engineering Library (BCEL) [10], Java reflection APIs [21], WEKA [2], and Monge-Elkan Similarity measure [1]. The current implementation of our approach supports an AspectJ compiler called ajc [11] Version 1.5 and Java 2 SDK v1.3 [22]. We next present the implementation details of the main components of our approach.

Aspect mining module. We have implemented the aspect mining module based on the Aspect Mining Tool (AMT) [13], an existing analysis framework developed for mining aspects as well as hidden concerns. The framework identifies aspects based on static analysis of programs and assumes that the programs pass type checking. AMT consists of two parts: the analyzer and the visualizer. The analyzer extracts aspects (crosscutting concerns) with the help of the ajc AspectJ compiler [11]. Our implementation uses only the analyzer to identify aspects from the source code given as input.

Join point identifier. The aspects output by the aspect mining module are fed as input to the join point identifier that we implemented. The join point identifier uses BCEL [10] and Java reflection APIs [21] to analyze the Java bytecode of the system under refactoring to identify the locations of aspects (i.e., the classes or methods that they belong to) and their types. The join point identifier also analyzes the instruction set of a method body to obtain

[4]The string aligner guarantees that the inferred pointcut expression matches the whole original set of intended join points.

the `before` and `after` code of a join point, in order to determine whether the refactored code would be realized as `before`, `after`, or `around` advice. The join points identified by the join point identifier are then fed as input to the clustering module.

Clustering module. The attribute identifier in the clustering module identifies the naming parts of a method as the modifier, return type, class name, method name, and arguments, being adopted from the Java language syntax. Based on the Java types, the module identifies the values of the naming parts and form the data set, being given as input to the WEKA clustering engine [2]. Then WEKA performs the clustering and outputs the clustered data set. The clustered data set is similar to the data set except that each item in the data section includes a cluster number that the item belongs to.

Inference module. The string alignment algorithm used by the string aligner is based on the similarity measure algorithms in Simmetrics [1], an open source similarity measure library. If the string aligner infers more than one inferred expression, the optimal one is selected based on the Monge-Elkan similarity measure [1]. This measure is the average of similarity measures of the pointcut expression with all join points. The pointcut testing framework in the module is based on APTE [3], an automated framework for testing pointcuts developed in our previous work. APTE is based on AJTE [25], an existing unit-testing framework without weaving. APTE outputs the set of join points matched by the inferred pointcut expression.

Our current implementation outputs aspects and their inferred pointcut expressions, which developers can use to form the refactored aspect code. In future work, we plan to extend the implementation to automatically synthesize aspects and pointcut expressions to produce aspect code.

5. Evaluation

This section presents the empirical results produced by applying our approach on selected subjects. We describe the objective, the subjects, and the process of the evaluation. We then present and discuss the empirical results.

5.1 Objective

The objective of the evaluation is to investigate the following questions:

1. Does clustering of join points and inference of pointcut expressions have a significant impact on the number of pointcut expressions to be used in the final refactored code?
2. Does the tool generate meaningful pointcut expressions?
3. Is there a real necessity for the usage of our tool?

5.2. Subjects

We have applied our tool on six applications from AspectJ benchmark suites (which can be obtained from `http://www.sable.mcgill.ca/benchmarks/`) and open source applications from SourceForge. Applications `Tetris`, `DCM`, and `StarJPool` have been selected from AspectJ benchmark suites. Applications `JARP`, `JHotDraw`, and `Tomcat` have been selected from SourceForge. `Tetris`, `DCM`, and `StarJPool` are a few of the benchmark programs collected from various sources on the world wide web in order to study the relative impact of several AspectJ language constructs on performance. Although these programs are AspectJ benchmarks, their sources provide equivalent Java versions. Hence the Java versions of `Tetris`, `DCM`, and `StarJPool` have been selected for our empirical study.

`Tomcat` is a servlet container used for Java Servlet and Java Server Pages technologies. `Tomcat` supports various large-scale and diverse range of applications. `JHotDraw` is a Java GUI framework for technical and structured Graphics. It has been developed as a design exercise but is already quite powerful. Its design relies heavily on some well-known design patterns. `JARP` is an auxiliary tool for Petri analysis that performs basic Petri net analysis and various other functions. All these tools have been widely adopted and are among the highly rated downloads in SourceForge. Hence these subjects have been selected to evaluate our approach.

5.3 Process

The source code from the applications is compiled and the lines of code, and the number of classes in each application are recorded in Column 2 of Table 1. Given the Java files of the application, the aspect mining module outputs the identified aspects, whose number is shown in Column 3. Then our approach identifies the join points for each aspect. The number of join points is recorded as the original number of pointcut expressions (Column 4). Since no clustering is performed at this stage, each join point has an expression associated with it and this expression is the original pointcut expression.

Our approach performs clustering on the identified join points and infers pointcut expressions. In the evaluation, we adopted the first option (described in Section 3.4) to deal with pointcut expressions of incorrect strength if any being detected. Since clustering is performed, the original number of pointcut expressions would be reduced. The reduced number of inferred pointcut expressions is shown in Column 5. The percentage reduction (Column 6) is calculated as the ratio of the difference in the numbers of pointcut expressions before and after applying our approach, to the

Table 1. Evaluation subjects and results

Applications	#Classes /LOC	# Aspects	Original #pointcut expressions	Reduced #pointcut expressions	Percentage reduction %	#Clusters	Avg #pointcut expressions per aspect/pointcut (original)	Avg #pointcut expressions per aspect/pointcut (reduced)
Tetris	17 / 1474	24	41	27	34.14	6	1.70	1.13
DCM	29 / 3384	62	351	196	44.15	46	5.66	3.16
StarJPool	191 / 16847	321	4860	2554	47.44	930	15.14	7.90
JARP	214 / 26790	513	12134	4375	63.94	1419	23.00	8.50
JHotDraw	398 / 28087	456	32661	9091	72.17	3096	71.62	19.93
Tomcat	455 / 45400	684	51108	9633	81.15	6608	74.72	14.08

original number of pointcut expressions. Column 7 shows the number of clusters that has more than one join point. The original and reduced average number of pointcut expressions per aspect (Columns 8 and 9) is calculated as the ratio of original and reduced number of pointcut expressions (Columns 4 and 5) to the number of aspects (Column 3).

5.4. Results

Table 1 shows the empirical results produced by applying our tool on the evaluation subjects described in Section 5.2 . Columns 1-3 show the applications used for evaluation, the number of class files and total lines of the Java source code in each application, and the number of aspects identified by AMT. Columns 4 and 5 show the number of pointcut expressions before and after using our tool. Column 6 shows the percentage reduction in the number of pointcut expressions achieved using our tool. Column 7 shows the number of clusters observed after using our tool. Columns 8 and 9 show the average number of pointcut expressions per aspect or pointcut, before and after using our tool, respectively.

From the table, we observe that with increase in the size of the source code, the size of the pointcut expressions also increases substantially. Combining the individual pointcut expressions with logical operators would be an inefficient solution. The result has shown efficient reduction ranging from 34% to 81% reduction in the total number of pointcut expressions. The larger an application is, the more substantial reduction our approach can achieve. Hence the probability of clustering many join points to a group is high. In the case of large applications, join points are concentrated in a larger number of locations in the source code. Furthermore, it is likely that a large number of join points share some common patterns in their naming conventions. Such naming patterns would allow effective clustering and inference of pointcut expressions.

As expected, we observe that the effectiveness of our approach is high when the join points for an aspect have a pattern in their naming convention suitable for inference. For example, the aspect `FigureAttributeConstant` in the `JHotDraw` application had 110 join points such as `getImage()`, `getPolygon()`, `getConstant(String)`, and `getColor()`. All the join points had a common pattern `get`. The inference engine generated the expression `get*(..)` to match all methods that started with `get` and had any number of arguments. In this example, we find that the aspect `FigureAttributeConstant` was found in methods that had a functionality of reading some kind of attribute. Since all the methods shared a common functionality, they also shared a pattern in their names, i.e., all the names start with `get`. Such naming conventions would allow our approach to generate effective results: a large number of join points can be grouped and a single expression is sufficient to represent all the join points.

Our approach performs clustering by determining common patterns in the name fields of join points. If a crosscutting concern consists of many join points, then it is likely that such a concern is concentrated in methods or classes that perform a similar kind of functionality. When the methods resemble in functionality, then it is likely that their naming conventions would resemble too. For example, consider the crosscutting concern `JComboBox` identified in the `JHotdraw` application. This concern had 13 join points such as `createIconkit()`, `createColor()`, `createTools()`, and `createTool()`. The names indicate that all the methods have a functionality of creating some kind of attribute. From this example, we found that all the join points had the common pattern `create` in their name fields. Our approach performed clustering of all the join points of this aspect that had the pattern `create` in their name fields. In this case, the expressions for 13 join points were replaced by a single expression. Thus the clustering has been effective in grouping join points.

When the methods resemble in functionality, it is likely that their naming conventions would resemble too. But this might not always be the case, and it is possible for two methods to be similar in functionality but have different naming conventions. Consider the aspect

InstructionHandle from the JHotDraw application. This aspect had join points such as readFromStream(), readHeader(), getMethodEntity(), and getStaticCounter(). Our approach clusters readFromStream(), readHeader() into one group, and getMethodEntity(), getStaticCounter() into another group. In this case, the join points under each group have a common pattern read or get in their names. But we found that both the patterns read and get reflect the same functionality of gathering details of some attribute. Although our results are effective, further reduction can be obtained if all the four join points had a common pattern in their names. For example, if the latter join points getMethodEntity() and getStaticCounter() have been named as readMethodEntity() and readStaticCounter(), respectively, the clustering would have generated a single expression rather than two, achieving further reduction. This case suggests that the output of our approach can suggest developers to refactor their naming conventions for methods to achieve higher reduction.

In summary, the results in Table 1 show that there is a significant reduction in the total number of pointcut expressions. The reduction ranges from 34% to 81%. If the straightforward approach is used, i.e., combining the pointcut expressions of each join point with a logical operator, then the average number of pointcut expressions is quite high as indicated in Table 1. For applications like JHotDraw and Tomcat, the average number of (original) pointcut expressions is quite high. Such complex pointcuts would be difficult to write, compile, and maintain.

Clustering join points based on a common characteristic and inferring pointcut expressions for each cluster effectively reduce the total number of pointcut expressions. Our approach clusters join points based on a common pattern in the split method-name parts (rather than individual characters in method names) of join points. This mechanism ensures that the inferred pointcut expressions are meaningful similar to expressions that developers would form. From the results, we observe that reducing the number of pointcut expressions and effectively generalizing the pointcut definition is necessary, and the results show that usage of our approach would help achieve better formulation of pointcuts.

6. Related Work

A variety of aspect mining techniques [13, 6, 23, 18, 8, 9, 20, 14, 7] have been developed. Our approach complements these existing aspect mining techniques used for aspect-oriented refactoring [17], serving as a post-processor of the results produced by these aspect-mining techniques.

Tonella and Ceccator [24] presented an assessment of the effects of migrating to aspect-oriented code. They conducted an empirical study on the object-oriented version and on the aspect-oriented version of the same system. They collected some metrics that quantify the maintenance and understanding effort. But they did not propose or provide an implementation of any techniques to perform refactoring. Our approach provides an automated way of performing the important phases of refactoring: mining aspects, identifying join points, and inferring pointcut expressions.

Binkley et al. [5] developed an approach to determine good and effective pointcuts in aspect-oriented refactoring. In this approach, when there are multiple join points and their corresponding multiple pointcut expressions, they define a new pointcut as the logical OR of the pointcut expressions formed for each individual join point. They do not cluster join points based on any common characteristic or infer effective pointcut expressions. Our approach clusters join points based on a common characteristic like their naming parts and infers pointcut expression from the clusters.

Binkley et al. [4] developed a semi-automated approach to support the migration from object-oriented code to aspect-oriented code. Their approach considers a source program with identified aspectual fragments as input and the approach produces a semantically equivalent aspect-oriented program. Their approach aims at reducing the overall cost associated with the refactoring activity. It requires manual refinement to generalize the pointcut definitions. It does not infer pointcut expressions from clustering join points. Our approach automates the whole process of refactoring including clustering join points to infer effective pointcut expressions.

Zhang et al. [26] developed the aspect refactoring verification tool that automatically verifies the refactored aspects against the original sources by checking for inequalities between them. Our approach provides an additional testing mechanism to test the strength of the pointcut expressions. The presence of this additional testing mechanism eliminates the need for a separate verification tool to test the inferred pointcut expressions.

7. Conclusion

Aspect-oriented refactoring involves mining aspects, identifying join points for the aspects, clustering join points, and grouping them under a pointcut definition. Manually performing this refactoring process is tedious due to increase in the number of crosscutting concerns as well the size of the source code. Our approach automatically identifies aspects and their join points, performs clustering based on the attributes of the identified join points, and infers a pointcut expression for each cluster of join points. The additional testing mechanism ensures that the inferred pointcut expressions are of correct strength. The experimental results show that a significant reduction (34% to 81%) in

the total number of pointcut expressions can be achieved using our approach. Effective refactoring can be conducted with this significant reduction in the number of pointcut expressions for aspects.

References

[1] SimMetrics: open source library of similarity metrics, 2006. http://www.dcs.shef.ac.uk/~sam/stringmetrics.html.

[2] WEKA 3: Data mining software in Java, 2006. http://www.cs.waikato.ac.nz/ml/weka/.

[3] P. Anbalagan and T. Xie. APTE: Automated pointcut testing for AspectJ programs. In *Proc. 2nd Workshop on Testing Aspect-Oriented Programs*, pages 27–32, 2006.

[4] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *Proc. International Conference on Software Maintenance*, pages 27–36, 2005.

[5] D. Binkley, M. Ceccato, M. Harman, and P. Tonella. Automated pointcut extraction. In *Proc. 1st Linking Aspect Technology and Evolution Workshop*, 2005.

[6] S. Breu and J. Krinke. Aspect mining using event traces. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 310–315, 2004.

[7] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proc. 21st IEEE International Conference on Automated Software Engineering*, pages 221–230, 2006.

[8] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. 20th IEEE International Conference on Software Maintenance*, pages 200–209, 2004.

[9] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *Proc. 13th International Workshop on Program Comprehension*, pages 13–22, 2005.

[10] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. http://jakarta.apache.org/bcel/.

[11] Eclipse. AspectJ compiler 1.5, May 2005. http://eclipse.org/aspectj/.

[12] R. E. Filman and T. Elrad. *Aspect Oriented Software Development*. Addison-Wesley Publishing Co., Inc., 2005.

[13] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *In Proc. Workshop on Advanced Separation of Concerns*, pages 220–242, 2001.

[14] A. Kellens and K. Mens. A survey of aspect mining tools and techniques. Technical Report 2005-08, INGI, UCL, Belgium, 2005.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[16] R. Laddad. *AspectJ in Action*. Manning, 2003.

[17] R. Laddad. *Aspect Oriented Refactoring*. Addison-Wesley, Sept 2006.

[18] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. 11th Working Conference on Reverse Engineering*, pages 132–141, 2004.

[19] R. Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[20] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, 2005.

[21] Sun Microsystems. *Java Reflection API*. Online manual, 2001.

[22] Sun Microsystems. Java 2 platform standard edition v1.4.2 API specification, 2003. http://java.sun.com/j2se/1.4.2/docs/api/.

[23] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proc. 11th Working Conference on Reverse Engineering*, pages 112–121, 2004.

[24] P. Tonella and M. Ceccato. Refactoring the aspectizable interfaces: an empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005.

[25] Y. Yamazaki, K. Sakurai, S. Matsuura, H. Masuhara, H. Hashiura, and S. Komiya. A unit testing framework for aspects without weaving. In *Proc. 1st Workshop on Testing Aspect-Oriented Programs*, March 2005.

[26] C. Zhang, H.-A. Jacobsen, J. Waterhouse, and A. Colyer. Aspect refactoring verifier. In *Proc. 1st Linking Aspect Technology and Evolution Workshop*, 2005.