Bridging the Gap between Aspect Mining and Refactoring

Isaac Yuen and Martin P. Robillard
School of Computer Science
McGill University
{iyuen|martin}@cs.mcgill.ca

ABSTRACT

Aspect-mining techniques help to identify crosscutting structure that could potentially be modularized through object-oriented (OO) or aspect-oriented refactoring (AO). This paper describes a case study in which we used aspect-mining techniques to identify and refactor crosscutting concerns using aspect-oriented programming. We observed that, in our case, there were many subtle variations in the implementation of the concerns that made them non-trivial to modularize with AO refactoring. In the end, we solved our modularization problem using traditional OO refactoring. We conclude that there exists an important gap between the identification of crosscutting concerns and the technologies available to mitigate the problem.

1. INTRODUCTION

Crosscutting concerns in a software system cause many problems, including scattered duplicate code. As a result, the system gradually becomes less maintainable. Aspect-oriented programming (AOP) proposes a solution to the crosscutting problem by supporting the modularization of crosscutting concerns into aspects. The availability of aspects suggests that it should be possible to incrementally refactor an existing object-oriented (OO) system into a more modularized AO equivalent. Nevertheless, this strategy poses at least three issues that must be addressed:

1. How to identify crosscutting concerns in a system;
2. How to determine if the concern code identified in step 1 is refactorable;
3. How to refactor crosscutting concerns into aspects, either manually or automatically.

Aspect mining addresses the first problem by identifying potential crosscutting concerns in a system using techniques such as *fan-in* analysis [15], lexical and dynamic analyses [6], and version history mining [4]. For crosscutting concerns that span many classes in a system, there exist tools that can help refactor the targeted code into an aspect automatically [2]. Such tools can reduce the effort required and the risks of introducing error typically associated with manual refactoring. However, between the identification of crosscutting concerns and the refactoring process, a developer must determine if the crosscutting concerns are refactorable. In particular, there are several issues that we must consider before applying refactoring, including:

- Does the AOP language support the extraction of the targeted crosscutting code?
- Can we deduce a pattern in the crosscutting code, such that we can apply a general pointcut that captures all relevant join points and produces a better modularized system?
- If we find variants in the crosscutting code that do not conform to the common patterns, what kinds of measure can we take to eliminate the discrepancy?

We conducted a case study to explore these questions by evaluating some crosscutting concerns in an open-source project. We employed *fan-in* analysis [15] as the aspect mining tool to help us locate the crosscutting concerns in the target system. We then manually inspected the results of the aspect-mining technique, and determined if there was an applicable AO refactoring [17] that could help us extract these crosscutting concerns into an aspect. We concluded that due to the variations in the code, it was difficult to choose a refactoring that neatly encapsulated the crosscutting concerns without sacrificing the readability and simplicity of the pointcut descriptor.

We resolved to refactor our crosscutting concerns by incrementally applying traditional OO refactoring such as *Extract Method* and *Pull Up Method* [8]. This strategy simplified the code and revealed some previously unseen relations between classes that participate in the implementation of the same crosscutting concern. We argue that comprehensive OO refactoring be a required preliminary step in assessing the applicability of AO refactoring.

2. CASE STUDY SETUP

The initial goal of our case study was to inspect crosscutting code and assess the difficulty in refactoring it into aspects. We chose an open-source Java project as our experimental target, and assumed AspectJ as the targeted AOP language for refactoring.

Workshop LATE '07 March 12-13, 2007 Vancouver, British Columbia, Canada

2.1 Target Systems

We applied our case study to an open-source project called Freemind[1]. FreeMind is a mind-mapping software written in Java. It is an editor for creating and navigating hierarchical diagrams. The version we used (0.8.0) consists of approximately 65,600 non-comment lines of code. FreeMind has over 5 years of development history and is currently maintained by 4-6 developers. FreeMind's medium user base (∼100,000 users) and its reputation[2] convince us that it is a representative example of a medium size Java project.

2.2 Locating crosscutting candidates

Fan-in analysis is an aspect-mining technique that identifies crosscutting concerns whose implementation consists of a large number of scattered invocations of specific functionality implemented by a method. The number of distinct calling methods gives the *fan-in* metric of the invoked method [14]. The FINT[3] Eclipse plug-in is a tool that supports *fan-in* analysis and navigation of crosscutting concerns, and was used as the aspect-mining tool in this case study. The results of FINT provided us with the fan-in values for all methods in the system. Since FINT does not provide information about the locations of the crosscutting calls, we manually inspected the call sites of each crosscutting candidate that has a *fan-in* value of at least 8 to determine if it forms a refactorable crosscutting concern.

2.3 Criteria for refactorable concerns

There is no formal guide to determine precisely what a refactorable crosscutting concern consists of. Therefore, we choose to look for a crosscutting concern that satisfies the following criteria:

Occurrence: A refactorable crosscutting concern should crosscut at least 8 different method bodies[4] in the system. In particular, we looked for clusters of at least 3 method invocations that can be found in all of the crosscutted method bodies.

Pattern: There should be a consistent pattern with respects to the order of invocations of these methods.

Absence of complex control structures: AspectJ does not have pointcut descriptors for control-flow such as loop, or switch statements [11]. Complex control structures also introduce irregularities among the crosscutting call sites. Therefore, we only considered crosscutting code that was not nested in control structure.

3. CASE STUDY RESULTS

We identified two refactorable crosscutting concerns using fan-in analysis. For each, we describe the nature of the concern's implementation, the challenges it poses to AO refactoring, our proposed solution, and our observations.

3.1 Overview of crosscutting candidates

The fan-in analysis of FreeMind reported a number of methods with fan-in higher than 100. However, we found that candidates that have high fan-in values are often not refactorable, either because they reside inside some control-flow structure (such as a `switch` statement), or because the locations of the call sites do not form any pattern. However, we located two groups of methods, with fan-in values of 32 and 49 respectively, that exhibit some recurring code patterns for refactoring. We outline the evaluation of each concern in the following sections.

3.2 Action Concern

Common pattern in crosscutting code. From the results of fan-in analysis, we noticed a group of methods that are invoked in the same pattern consistently, and whose fan-in value is equal or close to the other methods in the group. We identified that this group contains a code clone, and this clone is found in 32 different method bodies (see Table 1).

Figure 1 shows an example of this clone in `AddArrowLink Action.java`. In general, the clone consists of three method calls that are always in the same order: a call to `execute-Transaction()` is preceded by a call to `startTransaction()` and followed by a call to `endTransaction()`. In most cases, their call sites are adjacent to each other inside a method body. The fact that the clones are not located inside a loop or a branch makes the sequence of calls a good target for refactoring.

Crosscutting calls	Fan-in
ActionFactory.startTransaction(String))	33
ActionFactory.executeAction(ActionPair)	37
ActionFactory.endTransaction(String)	33
Intersection	32

Table 1: Fan-in values for methods of the *Action Control* concern

```
public void addLink(MindMapNode source, MindMapNode target)
{
    /* Begin clone */
    modeController.getActionFactory().
        startTransaction(String) getValue(NAME));
    modeController.getActionFactory().
        executeAction(getActionPair(source,target));
    modeController.getActionFactory().
        endTransaction((String) getValue(NAME));
    /* End clone */
}
```

Figure 1: *Action Control* concerns in `AddArrowLinkAction.java`

Challenges for AO refactoring. The code for this concern exhibits a *consistent behaviour* [13], and the method signatures reveal that it acts as a transactional control. There are two options to extract the crosscutting code into an aspect:

Option 1. Refactor the calls to `startTransaction()` *and* `endTransaction()` *into an 'around' advice of the containing method.*

This options requires that `startTransaction()` and `end-Transaction()` always be located at the beginning and end of the method body, which is not true for all cases. Figure 2 shows a variant that contains some initialization code before the clone.

Option 2. Refactor the `startTransaction()` *and* `endTrans-action()` *into an 'around' advice of* `executeAction()`. While the approach solves the problem created by the variant in

[1]http://freemind.sourceforge.net
[2]FreeMind was chosen as SourgeForge Project of the Month in February 2006.
[3]http://swerl.tudelft.nl/bin/view/AMR/FINT (v0.6)
[4]The default threshold fan-in value of FINT is 8

```
public void setEdgeColor(MindMapNode node, Color color) {
  try {
    doAction = createEdgeColorFormatAction(node, color);
    undoAction = createEdgeColorFormatAction(node,
            ((EdgeAdapter) node.getEdge()).getRealColor());
    /* Begin clone */
    controller.getActionFactory().
        startTransaction(this.getClass().getName());
    controller.getActionFactory().
        executeAction(new ActionPair(doAction, undoAction));
    controller.getActionFactory().
        endTransaction(this.getClass().getName());
    /* Ends clone */
  } catch (JAXBException e) {
        e.printStackTrace();
  }
}
```

Figure 2: Initialization code before `startTransaction()` in EdgeColorAction.java

Figure 2, it creates two new challenges. First, there are also variants of the clone in which `executeAction()` is not called right after the `startTransaction()` (see Figure 3). The fan-in analysis (Table 1) also shows that `executeAction()` has a higher fan-in value than the other two methods, and we noticed that there are several exceptions where `startTransaction()` and `endTransaction()` are not called along with `executeAction()`. A pointcut descriptor that accounts for these exceptions would be difficult to create.

```
public void setNodeText(MindMapNode selected, String newText){
    String oldText = selected.toString();

    try {}
        /* First part of the clone */
        c.getActionFactory().startTransaction(c.getText("edit_node"));

        EditNodeAction EditAction =
            c.getActionXmlFactory().createEditNodeAction();
        EditAction.setNode(c.getNodeID(selected));
        EditAction.setText(newText);
        EditNodeAction undoEditAction =
            c.getActionXmlFactory().createEditNodeAction();
        undoEditAction.setNode(c.getNodeID(selected));
        undoEditAction.setText(oldText);

        /* Second part of the clone */
        c.getActionFactory().executeAction(
            new ActionPair(EditAction, undoEditAction));
        c.getActionFactory().endTransaction(c.getText("edit_node"));
    } catch (JAXBException e) {
        e.printStackTrace();
    }
}
```

Figure 3: Clone variant in `EditAction.java`

Refactoring solution. The first step of refactoring must resolve the inconsistency found in Figures 2 and 3. Since in most cases all three calls in the clone are adjacent to each other, the best approach is to 'correct' the type of variant found in Figure 3 and change the locations of `startTransaction()` and `endTransaction()` to make them adjacent to the `executeAction()` call. The main challenge of the change is that we must verify that it does not change the behavior of the containing method body. In all cases of this crosscutting concern, we manually verified that this change was safe.

We then used the *Extract Method* technique to extract the clone in each class into a new method called `runTransaction` (see Figure 4).

After the extraction, we discovered an interesting core-

```
protected void runTransaction(ActionPair target,
                              String startName, String endName)
{
   modeController.getActionFactory().startTransaction(startName);
   modeController.getActionFactory().executeAction(target);
   modeController.getActionFactory().endTransaction(endName);
}
```

Figure 4: Extracted method `runTransaction()`

lation between the refactored classes. For instance, most of the classes that contain the clone are descendants of the `FreemindAction` class, even though the `FreemindAction` class does not contain the clone. Therefore we applied the *Extract Superclass* method repeatedly to extract the `runTransaction()` method into `FreemindAction`. This refactoring reduced the lines of code count by approximately 300.

3.3 XML Attribute Serialization concern

Common pattern in crosscutting code. From the results of the fan-in analysis, we noticed another group of method calls that represents another refactorable code clone. This group of code clones spans over 49 different method bodies (see Table 2). Figure 5 shows an example of this clone in the `AddIconActionTypeImpl.serializeAttributes()` method. In fact, every clone exclusively resides in the body of a `serializeAttributes()` method, and `serializeAttributes()` implements a method of the `XMLSerializable` interface. This implies that every class that is involved in this concern implements `XMLSerializable`. From this information we deduced that the clones belong to the XML Attribute Serialization concern.

```
public void serializeAttributes(XMLSerializer context)
        throws org.xml.sax.SAXException
{
    /* Clone begins */
    context.startAttribute("", "color");
    try {
        context.text(((java.lang.String)  _Color));
    } catch (java.lang.Exception e) {
        Util.handlePrintConversionException(this, e, context);
    }
    context.endAttribute();
    /* Clone ends */
    super.serializeAttributes(context);
}
```

Figure 5: An Example of the refactorable code clone in `AddIconActionTypeImpl.java`

Challenges with AO refactoring. The XML Attribute Serialization concern consists of an interface implementation and we considered it as a type of *role superimposition* concern [13]. There are two options to extract the crosscutting clones into an aspect:

Option 1. Refactor the clone into a 'before' advice of `serializeAttributes()`. Ideally, this option produces the most simplified and modularized implementation of the concern. However, there are variants of `serializeAttributes()` where this approach is not applicable. For instance, a variant may contain more than one instance of the clone, or the clone may reside in a control structure (e.g. inside an `if` branch or a `while` loop) (see Figure 6). Since the variants account for half of the method bodies that contain the clone, we would have to make some exclusions in the pointcut descriptor, and the concern would not be completely

Crosscutting calls	Fan-in
XMLSerializer.startAttribute(String, String)	49
XMLSerializer.text(String)	53
Util.handlePrintConversionException(Object, Exception, XMLSerializer)	50
XMLSerializer.endAttribute()	49
Intersection	49

Table 2: Fan-in values of *XML Attribute Serialization concern*

modularized in an aspect, hence defeating the purpose of the refactoring.

```
public void serializeAttributes(XMLSerializer context)
        throws org.xml.sax.SAXException
{   /* Clone begins */
    context.startAttribute("", "enabled");
    try {
        context.text(DatatypeConverter.printBoolean(_Enabled));
    } catch (java.lang.Exception e) {
        Util.handlePrintConversionException(this, e, context);
    }
    /* Clone ends */
    context.endAttribute();
    if (_Color!= null) {
    /* Clone begins */
        context.startAttribute("", "color");
        try {
            context.text(((java.lang.String) _Color));
        } catch (java.lang.Exception e) {
            Util.handlePrintConversionException(this, e, context);
        }
        context.endAttribute();
        /* Clone ends */
    }
    super.serializeAttributes(context);
}
```

Figure 6: A variant of `serializeAttributes()` that contains 2 clones.

Option 2. Refactor the clone into an 'around' advice of `XMLSerializer.text()`. Although this option can be generalized to every instance of the crosscutting call sites, the fan-in analysis (Table 2) shows that there are several situations in which `XMLSerializer.text()` is not called along with `XMLSerializer .startAttribute()` and `XMLSerializer.endAttribute()`. We must account for these exceptions in the pointcut descriptor.

Again, due to the irregularity of the crosscutting clones, neither option can provide a refactoring that completely and neatly encapsulates the crosscutting concern into an aspect. However, we could apply the same object-oriented refactoring strategy as the previous concern to eliminate the clones.

```
protected void serializeXMLAttribute(XMLSerializer context,
        String url, String local, String _text)
    throws org.xml.sax.SAXException {
    context.startAttribute("", local);
    try {
        context.text(_text);
    } catch (java.lang.Exception e) {
        Util.handlePrintConversionException(this, e, context);
    }
    context.endAttribute();
}
```

Figure 7: Extracted method `serializeXMLAttribute()` from the crosscutting code clones

Refactoring solution. . We applied the *Extract Method* technique to extract the clone into a new method called `serializeXMLAttribue()` (see Figure 7):

When we inspected the inheritance relationship between the classes that contain the crosscutting clone, we noticed that 33 of the 49 classes declaring our extracted method are descendants of the `XmlActionImpl` class. Therefore, we repeatedly applied the *Pull Up Method* to `serializeXMLAttribute()` until we reached the `XmlActionImpl` class. The refactored version reduced the line of code count by approximately 200 lines.

Using the inheritance graph, we also noticed that there are several classes that share the same prefix, such as `MenuActionBaseImpl`, `MenuCategoryBaseImpl`, and `MenuSubmenuTypeImpl`. Although these classes do not inherit from any other project class, we deduced that they are conceptually related and should be connected together through a superclass. We applied the *Extract Superclass* refactoring to refactor the `serializeXMLAttribue()` method in each class to a new superclass `AbstractMenuBaseTypeImpl`, and further eliminated approximately 90 lines of code.

4. DISCUSSION

The road from aspect-mining to aspect refactoring can be long and winding. Although aspect-mining techniques such as fan-in analysis were able to detect crosscutting functionality in our case study, only a small subset of the identified concerns could be refactored into aspects. One reason is that AOP languages such as AspectJ can only capture a small set of program execution points. Furthermore, AO transformations have many restrictions with respects to the structure and pattern of the crosscutting code, and can only be applied directly in certain circumstances [10]. This requirement of uniformity can rarely be expected from a large system, especially open source projects that have evolved over a long period of time. Developers may introduce changes to the code that disrupt the original structure or pattern and that breaks the regularity required for AO refactoring.

AO refactoring requires an initial stage of OO refactoring. In our case, crosscutting concerns were associated with code clones. There were several benefits to applying traditional OO transformations to refactor these clones. First, by extracting the code clones into a separate method, the intent of the clone becomes explicit, and the total code size is reduced. Secondly, for clone variants that cannot be directly refactored, we can study the discrepancies, which helps us to spot potential bugs in the system. Thirdly, we may be able to reveal some unseen relationships among the classes that share the same clone, and further refactor the code. Finally, untangling a concern is a major challenge in large-scale AO refactoring [2]. By repeatedly refactoring the code to a more concise form, our OO refactoring strategy may help us overcome this challenge.

5. RELATED WORKS

Aspect-mining techniques identify the crosscutting concerns in a system either statically or dynamically. Fan-in analysis [15] is a static aspect mining technique that determines the scatteredness of code by identifying methods that are called in many places in the project. Lexical-based analysis derives crosscutting concerns by grouping program elements based on their lexical representations [6]. A version-history based approach by Breu and Zimmermann [4] analyzes the addition and evolution of program elements and correlates them with the author and timestamp data from the version control history. This approach is more scalable to large projects since the precision of the crosscutting concerns increases with the project size and history. A case study by Bruntink et al. [5] applies several clone detection techniques to an industrial C application and analyzes their effectiveness in finding the crosscutting concerns. Dynamic aspect mining [3] depends on run-time program behavior to identify recurring execution patterns that can be classified as a concern. Tonella and Ceccato apply concept analysis [18] to analyze how execution traces are related to class methods and identify related methods as a crosscutting concern.

There are several case studies that evaluates the applicability of AO refactoring. Monteiro et al. [16] and Hannemann et al. [9] separately conducted studies that applied AO refactorings incrementally on small code examples that use design patterns such as the *Observer Pattern*. Marin also illustrated the AO refactoring approach on JHotDraw project and completely extracted the Undo concern into an aspect [12]. However, the crosscutting concerns identified in these projects do not exhibit the irregularities in our FreeMind examples. Coyler et al. [7] also conducted a case study that seperated the EJB support from an large-scale application server using AO refactoring.

Automated AO refactoring remains a challenge. Binkley at al. [1] [2] created a semi-automated approach to refactor object-oriented program elements into aspects. Currently, the prototype lacks syntactic checks and oftens provide erronous refactored code and unreadable pointcuts that can be improved. Hannemann et al. [10] developed another semi-automated approach called role-based refactoring. However, the target code must conform to certain design patterns to enable the refactoring and is not flexible for general use.

6. CONCLUSION

We described our efforts to locate refactorable crosscutting concerns in an open-source project using an aspect-mining technique, and to refactor the identified code to mitigate their crosscutting nature. The concerns that could be refactored were rare, and we discovered that we could not find an appropriate AO refactoring approach that could produce a simpler and more modularized code, due to the variations in the structure. We resorted to a traditional OO refactoring strategy, and discovered some hidden relationships between classes that helped us further simplify the code as a step that could facilitate future AO migration. We conclude that there exists an important gap between the identification of crosscutting concerns and the technologies available to mitigate the problem.

7. ACKNOWLEDGEMENT

This work was supported by the Fond Québécois de la recherche sur la nature et les technologies (FQRNT).

8. REFERENCES

[1] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 27–36, 2005.

[2] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.

[3] S. Breu and J. Krinke. Aspect mining using event traces. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 310–315, 2004.

[4] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of the 21th IEEE international conference on Automated software engineering*, pages 221–230, 2006.

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

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

[7] A. Colyer and A. Clement. Large-scale aosd for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, 2004.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[9] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, 2002.

[10] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, 2005.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, 2004.

[12] M. Marin. Refactoring JHotDraw's Undo concern to AspectJ. In *Proceedings of the First Workshop on Aspect Reverse Engineering (WARE).*, 2004.

[13] M. Marin, L. Moonen, and A. van Deursen. A classification of crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, 2005.

[14] M. Marin, L. Moonen, and A. van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Proceedings of the 13th*

IEEE Working Conference on Reverse Engineering, pages 29–38, 2006.

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

[16] M. P. Monteiro and J. M. Fernandes. Refactoring a java code base to aspectj: An illustrative example. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 17–26, 2005.

[17] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, 2005.

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