Refactoring to Aspects – an Interactive Approach

Jan Hannemann, Thomas Fritz and Gail C. Murphy

University of British Columbia

Abstract

Current refactorings for restructuring existing software systems preserve the behavior of the software. Aspect-oriented programming constructs make existing refactorings more complex, and introduce the potential for new kinds of refactorings that target entire concerns. In many cases, behaviour preservation may be neither possible nor desirable. In this position paper, we propose an approach to refactoring aspects into a system that actively involves a developer in a dialogue with the refactoring tool. We are exploring this approach by developing two Eclipse plug-ins: one which bases the refactoring on descriptions of a concern in the code; the other which bases the refactoring on a target aspect structure.

1 Introduction

Software developers refactor source code primarily to make subsequent changes to the code easier to perform. The term refactoring has become synonymous with changes to the structure of the code that retain the behaviour of the system (e.g., [1, 3, 6]). Such refactorings are playing a useful role in object-oriented programming: they are an integral part of extreme programming, and many IDEs, including Eclipse, have built-in support.

Languages that support aspect-oriented programming [5] introduce new programming structures to capture crosscutting concerns. For example, the AspectJ language includes structures such as the aspect, the pointcut, and advice. These new structures introduce the possibility for new refactorings. For instance, one can define refactorings for aspect structures that are similar to existing object-oriented refactorings.

Although such refactorings can help a developer make detailed changes to the software structure, they have two limitations. First, it can be difficult to keep the aspect structures meaningful as object-oriented refactorings are applied. Consider, for example, the program and pointcut shown in Figure 1. Even a relatively simple refactoring, such as inlining the `isPriorityOK` method (see Figure 2), complicates behavior preservation since the join point for the `priorityCheck` pointcut ceases to be available. Although it is not clear how often the inlining refactoring is used, it exists and raises an issue for AOP refactoring tools.

```
int getStatus() {
  ...
  if (isPriorityOK()) {
    ...
  }
  ...
}

boolean isPriorityOK() {
  return _priority >= MIN_PRIORITY;
}

pointcut priorityCheck():
  call(boolean isPriorityOK());
```

Figure 1. A simple program and pointcut

```
int getStatus() {
  ...
  if (_priority >= MIN_PRIORITY) {
    ...
  }
  ...
}
```

Figure 2. After applying the inlineMethod refactoring the necessary join point is lost and PC is broken

Second, since most aspect structures help capture *crosscutting* concerns, it can be difficult to refactor a concern by applying a few localized refactoring operations. For instance, consider the change monitoring example presented in the AspectJ documentation (`www.eclipse.org/aspectj`). In the example, the concern of refreshing a display when a figure on that display moves is encapsulated in an aspect (Figure 3), which includes state (the `dirty` variable), a helper method (`testAndClear`), a pointcut that names methods in multiple classes, and advice. Moving and restructuring the code that appears in the MoveTracking aspect would require multiple low-level refactorings, such as "include method in pointcut", "move state", and so on.

```
aspect MoveTracking {

  private static boolean dirty = false;

  public static boolean testAndClear()
    {…}

  pointcut move():
    call(void FigureElement.setXY(…))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point))
    || call(void Point.setX(int))
    || call(void Point.setY(int));

  after() returning: move() { … }
}
```

Figure 3. MoveTracking Aspect

We are investigating an approach to introducing new aspect structures into object- and aspect-oriented systems that is based on engaging the developer in a dialogue. The intent of the dialogue is to resolve the design and implementation choices that occur for capturing crosscutting concerns as aspects. The refactoring approaches we are investigating are not meaning-preserving because of the range of choices available.

In this paper, we briefly describe two dialogue-based approaches. One approach focuses on the use of a crosscutting concern description captured with a concern elaboration tool, and the subsequent use of that description as a basis for refactoring. The other approach is based on describing an aspect structure, in our case an aspect-description of a design pattern, to use as a target for the refactoring. We are implementing each of these approaches as an Eclipse plug-in.

2 Refactoring from Concern Descriptions

The FEAT concern elaboration tool (`www.cs.ubc.ca/labs/spl/projects/feat`) helps a developer describe the implementation of a concern that is scattered across the source code of a system as a concern graph [7]. The information in a concern graph consists of the program elements (e.g., classes, methods, and fields), and the relationships between those elements that are related to the concern. As an example, a concern graph that was developed to describe the auto-saving feature of the JEdit system (`www.jedit.org`) contains the information that the `Autosave.actionPerformed` method calls the `Buffer.autosave` method.

We can use the information in a concern graph description of a concern as a basis for refactoring the concern into an aspect-oriented structure. Specifically, we are investigating the refactoring of a concern graph into an AspectJ aspect.

Our approach consists of three steps. In the first step, the developer is asked to specify which elements and relationships from the concern graph should be included in the aspect. Figure 4 shows a page in an Eclipse plug-in wizard that helps with this step. This page allows a developer to specify that the calling of the `Buffer.autosave` method by the `Autosave.actionPerformed` method should be encapsulated in the aspect, but that the `actionPerformed` method itself should be retained in the `Autosave` class as the developer may know that the method also includes code related to other concerns. In other cases, the developer may choose to refactor the entire program element, such as moving the entire `actionPerformed` method into the aspect.

The second step consists of planning the refactoring. The operation of refactoring the concern graph into an aspect consists of many sub-refactorings:

- Refactoring an entire class moves that class into a new package created for the aspect.
- Refactoring a method or field causes the movement of the field or method into the aspect, with a static crosscutting statement used to (re)introduce the method or field into the code from which it was removed.
- Refactoring a *simple* field access or method call relationship causes the involved code to be moved into an advice structure within the aspect. A hook method is introduced into the original code to provide an appropriate join point.
- Refactoring a *difficult* field access or method call relationship, such as one that is enclosed in a control structure or that involves changing a local variable, causes the plug-in tool to engage the developer in a dialogue. In some cases, the dialogue consists of the tool suggesting an automated refactoring, such as using the object-oriented `extract method` refactoring to insert a needed hook method, and the subsequent introduction of advice into the aspect (Figure 5). In other cases, the tool can provide the devel-

oper with information about the enclosing control structures, but can suggest restructurings that would need to be manually applied by the developer.

In the third step, the tool applies the refactoring plans worked out with the developer. When a refactoring can not be fully-automated, the tool leaves TODO notes in the code.

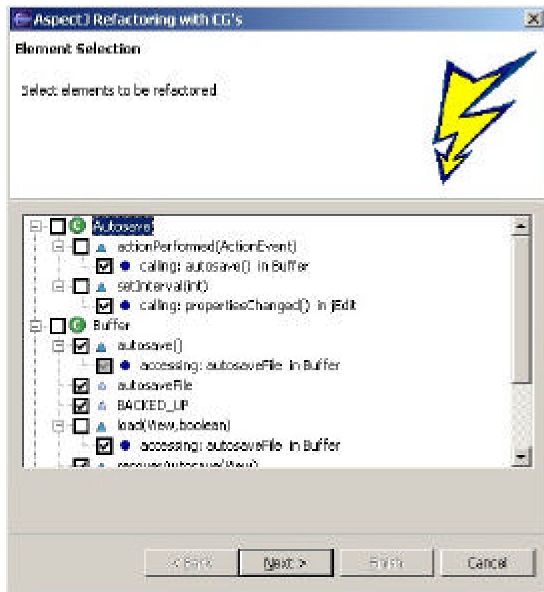

Figure 4. Selection of elements and relations to be refactored based on the Autosave Concern Graph

Our plug-in to support this approach is layered on the FEAT plug-in. We rely upon FEAT for most of the information about how and where program elements interact in the code base. FEAT extracts this information from the JDT. Since in addition to the inter-program element information provided by FEAT, we also need intra-program element information, such as whether a call statement is embedded in a control structure, we also interact with the JDT directly. We abstract the information from FEAT and the JDT into a refactoring database that assures properties that need to hold for elements to be refactored:

- If a field is to be refactored, all statements accessing the field must also be refactored.
- If a statement accessing a field is refactored, we must also refactor the field (and hence all statements accessing the field).
- If a method is to be refactored, all statements that call the method must be refactored.

```
public void actionPerformed(..) {
  ...
  Buffer[] bufferArray =
    jEdit.getBuffers();
  for(int i=0;i<bufferArray.length;i++){
     bufferArray[i].autosave();
   }
   ...
}
```

(a) original code: calling of autosave() in actionPerformed(..) is enclosed in a for-statement

```
public void actionPerformed(..) {
  ...
  Buffer[] bufferArray =
    jEdit.getBuffers();
  hookAutosave(bufferArray);
  ...
}

private void hookAutosave(Buffer[] ba){}

after(Buffer[] ba):
  call(* *.hookAutosave(Buffer[])
  && args(ba){
    for(int i=0;i<ba.length;i++){
      ba[i].autosave();
    }
```

(b) hook method and created advice

Figure 5. Refactoring of a call statement

- If a class is to be refactored, all program elements declared by the class must also be refactored.

Our plug-in also consist of the wizard that interacts with the developer to plan the refactoring, and the actual refactoring tool, which moves program elements from the original code into a new privileged aspect. To date, we have implemented the database portion of the plug-in and the wizard. We are now working on the actual refactoring tool.

3 Refactoring Design Patterns

In our second approach, rather than working from a largely source-based description of a crosscutting concern, we are refactoring in the presence of an aspect-oriented target. Specifically, we are focusing on the refactoring of object-oriented implementations of GoF design patterns [2] into aspect-oriented forms. These design patterns are a reasonable target as a previous study showed that aspect-

oriented implementations of the patterns could exhibit a number of desirable properties [4].

To better understand the challenges involved in this kind of refactoring, we picked several design patterns in the JHotDraw graphical editor framework (`www.jhotdraw.org`) and replaced the object-oriented implementations with aspect-oriented versions from the earlier study. We found that during this process we were faced with implementation alternatives, and that there were cases in which it was not desirable to preserve the original behaviour.

For example, in the implementation of the observer pattern between `DrawingView` and `FigureSelectionListener` in JHotDraw, the original code notifies listeners after calls to `addSelection(Figure)`, but not after calls to `addSelectionAll(Figure[])`. The latter method calls `addSelection(Figure)` to ensure proper updates. A straightforward aspect-oriented version of the design pattern updates after both kinds of calls, resulting in an additional call to listeners to update after adding an array of figures to the selection. Although a user of the program may not notice a difference, this refactoring is not behaviour preserving. However, the aspect-oriented version is safer because it captures the intent of the pattern: listeners should be informed of the addition of figures. If a developer changes the implementation of `addSelectionAll(Figure[])` in the future to not rely upon `addSelection(Figure)`, in the solely object-oriented version, listeners may not be informed. The aspect-oriented captures the intent directly, ensuring that listeners will be informed.

To help guide a tool in these choices, we are investigating suitable dialogue interaction between the developer and the tool. Our approach, which we are developing as an Eclipse plug-in, has two fundamental parts. First, we need a means of expressing the target aspect structures, in our case the aspect-oriented design patterns, that are sufficiently intentional to be useful to the average developer, but expressive enough to be allow for complex refactorings. To allow for both high-level (intentional) and low-level (expressive) refactorings, we plan to employ a composite approach to refactoring that allows developers to create refactorings that are compositions of other refactorings. Developers can then define and apply a complex high-level refactoring such as "replace this object-oriented Observer design pattern with this aspect-oriented version", while still being able to refine a refactoring when necessary.

Second, the dialogues used by the tool must be concise and helpful. Too many dialogues might make the refactoring process tedious, while too few might exclude the developer from an important decision. We are investigating two approaches for the dialogues: impact analysis of the different choices in which the tool provides the developer with information about the consequences of each choice, and learning approaches in which the tool "learns" from prior decisions made by the developer. Figure 6 shows a portion of a sample dialogue with the developer in which an object-oriented instance of the observer pattern is replaced with an aspect-oriented version. In this portion of the refactoring, participant classes will no longer need to implement a Subject interface. The tool can detect this situation and make suggestions.

An essential step in the development of this plug-in is the determination of an appropriate infrastructure to access the necessary information about the static structure of the program to be refactored. Since accessing the abstract syntax trees in the JDT via visitors is rather inefficient, we are creating an in-memory database of the critical AST elements and their relationships through a walk of the syntax trees. Information about the aspects affecting the system will be derived from the AJDT plug-in for Eclipse, which provides an API for querying the aspect structure of AspectJ programs.

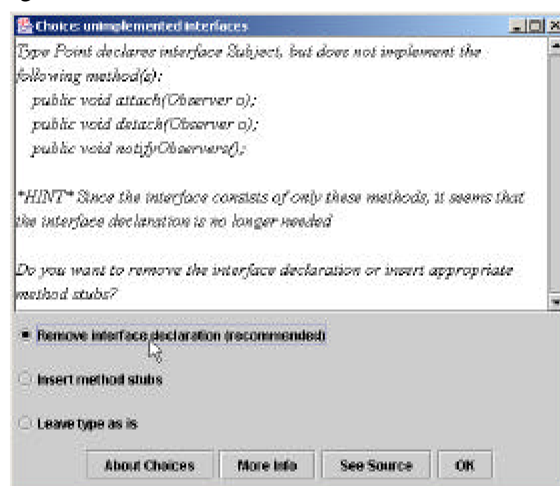

Figure 6. A sample dialogue

4 Concluding Remarks

Aspect-oriented programming constructs make new kinds of refactorings possible. The nature of these constructs suggests a need to reevaluate the strong link between refactoring and behaviour preservation. For refactorings that target non-aspect structures in a program, such as existing object-oriented refactorings, preserving behaviour is usually desirable. It should be possible to restructure the base code independent of the aspect structures influencing it. In this case, tool support can help ensure behaviour preservation. However, this property may not always be possible. For refactorings that target aspect-oriented structures, capturing the intent of the refactoring may be more important than behaviour preservation.

As soon as the behavior of a program can not be preserved, developer interaction is required to evaluate the consequences of the change. We believe that a focused dialogue between the developer and the tool is an effective way to perform this interaction. We are investigating two approaches to refactoring that result in two different kinds of dialogues. In one approach, the refactoring is described solely in terms of the pieces of source to restructure. The dialogue in this case focuses the developer on low-level, localized decisions, such as how to restructure an existing method to expose an appropriate join point. In the other approach, the refactoring is described at a higher-level in terms of a target aspect. The dialogue in this case focuses the developer on more design-level, aspect-oriented decisions, such as the scope of pointcuts. By considering both approaches, we hope to investigate when each of the approaches is most effective, and what common infrastructure is needed to support each refactoring approach.

Acknowledgements

This work was supported in part by NSERC and in part by IBM. Thanks to Gregor Kiczales for his valuable comments on earlier drafts of the paper.

About the Authors

Jan Hannemann is a doctoral candidate at the University of British Columbia. His thesis work and research interests include design patterns, dialogue-based approaches to refactoring and refactoring of aspect-oriented systems. His email address is jan at cs.ubc.ca.

Thomas Fritz is a student of Computer Science at the Ludwig-Maximilians-University Munich and is currently working as a research student for Gail Murphy at the University of British Columbia. His research interests include software engineering and distributed systems. His email address is fritzt at informatik.uni-muenchen.de.

Gail C. Murphy is an Associate Professor at the University of British Columbia. Her research interests include software evolution, aspect-oriented software development, and empirical evaluation techniques. Her email address is murphy at cs.ubc.ca.

References

[1] M. Fowler. Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1999.

[2] E. Gamma et. al. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994

[3] W.G. Griswold. Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, Technical Report No. 91-04-04, 1991.

[4] J. Hannemann, G. Kiczales. Design Pattern Implementation in Java and AspectJ. In Proc. of OOPSLA, pages 161-173, 2002

[5] G. Kiczales et. al. Aspect-Oriented Programming. In *Proc. of the ESEC/FSE*, LNCS 1241, pages 220-242, 1997.

[6] W. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[7] M.P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In Proc. of *ICSE, pages, 2002.*