IMPROVING SOFTWARE MODULARITY THROUGH CROSSCUTTING CONCERN EXTRACTION

by

Isaac Yuen

School of Computer Science
McGill University, Montreal

April 2009

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Aspect-oriented programming (AOP) is a programming paradigm for improving the modularity of software systems by localizing crosscutting concerns in the system into aspects. Aspect-oriented refactorings extend AOP to legacy systems, by identifying and encapsulating existing crosscutting concerns through aspect-mining (discovery of crosscutting concerns) and aspect refactoring (semantic-preserving code transformation to extract the crosscutting code into aspects). However, not all the data obtained from aspect-mining corresponds to crosscutting concerns, and existing aspect languages may not be capable of refactoring all crosscutting concerns into aspects. In this thesis, we describe an approach for extracting crosscutting concerns in a system to a form that is suitable for refactoring. This process includes identifying the presence of crosscutting code clusters in aspect-mining results; assessing if the concerns should be extracted using various metrics; and performing code transformation to extract the crosscutting clusters into standalone methods with a common method signature and parameters. The work also describes the *ConcernExtractor*, a software tool that implements the concern cluster extraction technique. We apply *ConcernExtractor* to assess and extract the crosscutting concerns in existing systems to evaluate the prevalence of crosscutting concerns that are refactorable, and the applicability of our approach for generating aspect refactoring opportunities.

Résumé

La programmation orientée-aspect (POA) est un paradigme ayant pour but d'améliorer la modularité du logiciel en localisant les préoccupations éparpillées dans des aspects. La refactorisation orientée-aspect étend les avantages de la POA par l'identification et l'encapsulation des préoccupations existantes à l'aide du forage d'aspects (aspect mining), et par leur refactorisation en aspects à l'aide de transformations de code. Cependant, certains résultats du forage d'aspects ne correspondent pas à des préoccupations éparpillées, et les languages aspects existants ne supportent pas la refactorisation de tous les préoccupations en aspects. Cette thèse décrit une approche pour extraire les préoccupations éparpillées dans une forme qui se prête à la refactorisation. Le processus inclue l'identi-fication des préoccupations refactorisables parmis les résultats de forage d'aspects, l'évaluation de la valeur de l'extraction potentielle, et l'extraction proprement dite à l'aide de transformations de code. La thèse décrit aussi ConcernExtractor, l'outil que nous avons réalisé pour supporter cette approche. Nous avons appliqué ConcernExtractor pour évaluer l'approche sur plusieurs systèmes existants.

Acknowledgments

The journey of undertaking and completing this work is a daunting and self experience for me. On numerous occasions I was on the verge of giving up, thinking that my work was not good enough, or I did not meet the expectations of my mentors, family, and friends. Yet it was their continuous encouragements, and prayers throughout that have sustained me. Words alone cannot fully express my gratitude to them all.

I am most indebted my supervisor Martin Robillard, from whom I have learned the integrity and meticulosity to scientific research, which I can only hope to emulate in my career. I am thankful for his guidance, and challenges that taught me to become vigilant to the quality of my work and research. However, I am most grateful for his continuous encouragements and patience with me. “The great teacher inspires” – so thank you Martin, for keeping me from stopping to believe in myself.

The sustenance of my family is the most vital force that compels me to pursue my goal and not to give and I just want to tell them, “Dad, Mom, and Sis, thank you for your prayers and your patience with me. Now that I’m finally done, I just hope that you would be proud of me”.

Last but not least, thank you for all my family and friends whom never ceased to offer their encouragements, occasional rebukes ☺, and prayers. Without them, I would have long forgotten my “mission” and settled for complacency.

And thank you Lord Jesus. Your grace is sufficient, and Your power is made perfect in my weaknesses.

Contents

List of Figures

List of Tables

Listings

Chapter 1
Introduction

Object-oriented programming (OOP) is probably the most popular programming paradigm of this generation and is the principal methodology for designing and implementing software systems today. By modeling each concern or functionality into a separate module, the OOP intends to bring better modularity to the design of the system. However, OOP is not without its imperfections. Since OOP implicitly mandates that a functionality be modularized in only one dimension (objects), many kinds of concerns that do not align with that dimension become scattered across many modules and tangled with one another – the phenomenon that is called the *tyranny of the dominant decomposition* [36]. These types of scattered and tangled concerns are described as *crosscutting concerns*.

Crosscutting concerns (CCC) describe computational units that provide similar functionalities, but cannot be abstracted into a standalone module due to the limitations of the programming language. One of the much-cited examples of crosscutting concern is the logging functionality in Apache Tomcat server system.

Figure 1.1 shows a mapping of the classes in the JBoss Web API networking package[1]. In the graph, each column represents a file that contains one or more classes in the package, and each line of code in the file is represented by a row of pixels in the column. The highlighted area in the diagram represents the code that is related to URL management.

[1] `org.apache.tomcat.util.net` package

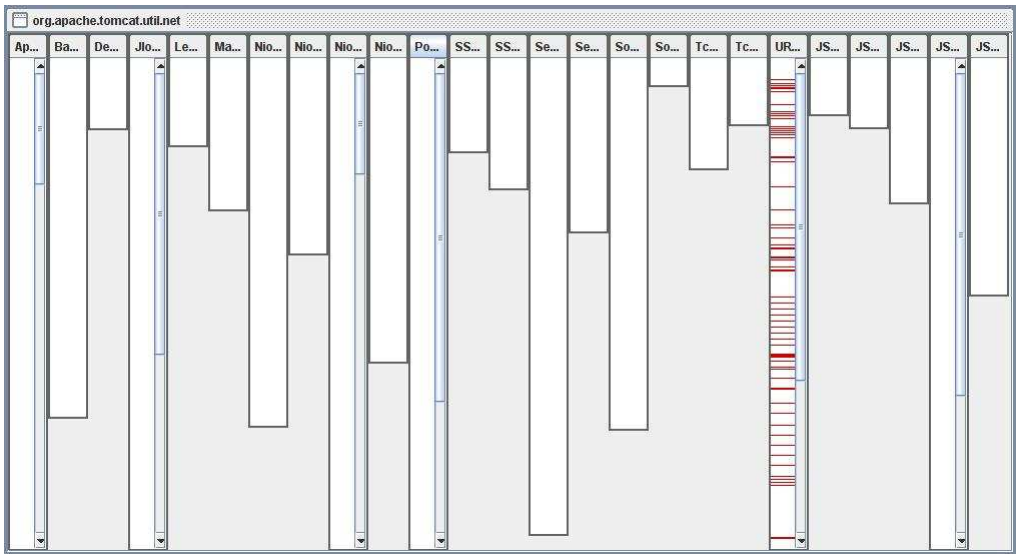

Figure 1.1: Location of URL management code in *org.apache.tomcat.util.net* package

This URL management concern, as the diagram shows, is well modularized inside a single class.

However, not every concern in the package can be encapsulated in a single class. Figure 1.2 shows the locations of the logging-related function calls in the system, and one can observe that the calls are dispersed among many classes, which is a classic example of a *crosscutting concern* in a system. The presence of crosscutting concerns in a system means that some functionalities gets *scattered* across different components, and become *tangled* with other components. As a result, the system not only become less readable and traceable, but individual components are less reusable because they include functionality that may not apply to the context in which the code is reused.

Aspect-oriented programming (AOP) provides a solution to the crosscutting problem by supporting the modularization of crosscutting concerns in a new construct called *aspect* [21]. From the OOP's perspective, an aspect can be understood as a special class with functions that do not need to be directly referenced in another class in order to be invoked. Aspects allow a programmer to localize the crosscutting concerns in the system

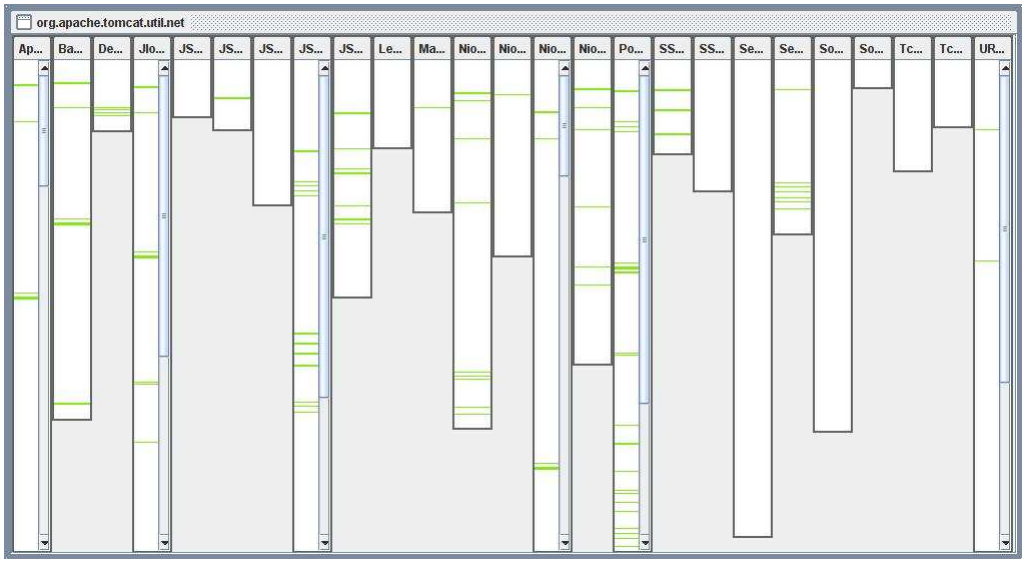

Figure 1.2: Location of logging code in *org.apache.tomcat.util.net* package

from the core modules, and remove *scattering* (a common code fragment dispersed at multiple locations) and *tangling* (a code fragment that serves different unrelated functionality) problems in the code.

Among modern aspect languages, AspectJ[2] is the most widely known and studied. Implemented as an AOP extension of Java, AspectJ introduces many new programming constructs, such as, *pointcuts* and *advice*, which became standard construct for most aspect languages.

A *pointcut* is an expression for specifying a set of *joinpoints*, which are well-defined points in the execution flow of a program. After specifying the pointcuts to locate the targeted joinpoints in the program, additional code can be applied before or after the joinpoints to introduce additional behavior. These additional code fragments are defined in a special type of method body called *advice*. Both *advices* and *pointcuts* are declared in an *aspect*. Typical joinpoints in OOP systems are method invocations or field accesses.

Listing 1.1 shows an example of an aspect class in AspectJ that implements clipping

[2]AspectJ. See http://www.eclipse.org/aspectj

```
aspect BoundaryClippingAspect {

  pointcut checkLineBound(int x1, int y1, int x2, int y2):
  (call(void Graphics.drawLine(int, int, int, int)) &&
  args(x1, y1, x2, y2))

  pointcut checkRectBound(int x1, int y1, int width, int height):
  (call(void Graphics.drawRect(int, int, int, int)) &&
  args(x1, y1, width, height));

  void around(int x1, int y1, int x2, int y2): checkLineBound(x1, y1, x2, y2)
  {
    if ( x1 < MIN_X )
      x1 = MIN_X;
    else if ( x1 > MAX_X )
      x1 = MAX_X;
    if ( x2 > MAX_X )
      x2 = MAX_X;
    else if ( x2 < MIN_X )
      x2 = MIN_X;
    if ( y1 < MIN_Y )
      y1 = MIN_Y;
    else if ( y1 > MAX_Y )
      y1 = MAX_y;
    if ( y2 > MAX_Y )
      y2 = MAX_y;
    else ( y2 < MIN_Y )
      y2 = MIN_y;

    // Call the drawLine method with the clipped parameters
    proceed(x1, y1, x2, y2);
  }

  void around(int x1, int y1, int width, int height):
  checkRectBound(x1, y1, width, height)
  {
    if ( x1 < MIN_X )
      x1 = MIN_X;
    if ( x2+width > MAX_X )
      width = MAX_X - x1;
    if ( y1 < MIN_Y )
      y1 = MIN_Y;
    if ( y2+height > MAX_Y )
      height = MAX_y - y1;

    // Call the drawRect method with the clipped parameters
    proceed(x1, y1, width, height);
  }
}
```

Listing 1.1: An AspectJ example for clipping the x,y co-ordinates of operations that draw lines and rectangle on a canvas

in the `Graphics.draw` operations. The intention is to clip the arguments of the functions if the method attempts to draw beyond the pre-defined boundaries. However, instead of adding a pre-condition check before every `drawLine` or `drawRect` call, as the typical OOP practice dictates, the clipping operation is modularized into an aspect class. In the aspect definition, two pointcuts are created to intercept any call to the `drawLine` and `drawRect` methods. The advices then check if the arguments of the methods are within the pre-defined boundaries, and if not, modify the value of the arguments to achieve the clipping effect.

The availability of AOP technology suggests that it should be possible to incrementally refactor an existing object-oriented (OO) system into a more modularized AO equivalent. However, refactoring case studies [6, 9] show that manual aspect refactoring is time-consuming and often not scalable in large applications: it is simply too onerous to manually inspect code to find cross-cutting concerns, and to manually transform the code to mitigate them. The scalability problem produced two subareas in AOP research: *aspect-mining* (automated detection of CCC in an OO systems) and *tool-based AO refactoring* (automated code transformation into aspect). However, the feasibility of combining the techniques from both domains to reduce the efforts in the AO migration process remains a challenge, due to the following reasons:

- Aspect-mining techniques are not precise and only collect programming elements that exhibit crosscutting attributes. It often requires human judgment to determine if they form a distinct crosscutting concern, and how to refactor it into an aspect.
- The code snippets that belong to a common crosscutting concern rarely have the same code structure. For instance, the sequence of executions may differ between each code snippet, or the execution maybe not be consecutive [38], making it difficult to automate code refactorings.
- Existing AOP languages such as AspectJ impose constraints that hinder the refactoring into aspects. For instance, AspectJ semantics does not provide access to local

variables[3], and it is not possible to encapsulate two identical method calls that are declared in the same method body with one pointcut expression [1].

1.1 Motivation

Although *crosscutting concerns* imply functionalities that are not modularizable using classes or modules, there is no criteria that explicitly determine if some code snippets constitute a crosscutting concern and should be implemented in aspects.

Marin et al. classified crosscutting concerns into different *sorts* based on their intents and their associated AO refactoring strategy [25]. In general, *consistent behavior* — the consistent calling of a method from several points in the program, is the most prevalent crosscutting sort. A concern that belongs to the *consistent behavior* sort appears in three formats:

1. A method invocation independent of the context of its caller, such as logging, tracing.

2. Variations of the `begin-/end-` coding pattern that execute the code between the `begin-` and `end-` methods in a different context. Examples of such patterns include the *Java* `java.util.concurrent.locks.Lock` interface (the `lock()` and `unlock()` methods)[4] or the *Batch Edit* concern found in jEdit (see Listing 1.2).

3. A sequence of method invocation statements that are consistently executed sequentially at multiple locations throughout the system.

We deem that it is most useful to refactor the third class of crosscutting concern, namely, the multiple methods that are repeatedly invoked together throughout the system, because the "clustering" of these methods is a better indication of a non-trivial concern

[3] Aspect FAQs. See http://www.eclipse.org/aspectj/doc/released/faq.php
[4] Java Lock interface API. See http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/Lock.html

```
public void writeNewStyleItem(String name, Map props) {
  // ... formatting code is omitted
  try {
    buffer.beginCompoundEdit();

    if (i == -1) {
      start = area.getCaretPosition();
    } else {
      start = i;
      int closingBracket = bufferText.indexOf("\}", i);
      if (closingBracket != -1) {
        buffer.remove(start, closingBracket - start + 1);
      }
    }
    buffer.insert(start, text);
  } finally {
    buffer.endCompoundEdit();
  }
}
```

Listing 1.2: An example of *begin/end* pattern in `JEditTextWriter` class, *jEdit 4.2*

that is more granular than a single function. By grouping and refactoring these concerns, we can improve the modularity of the system. Moreover, since the concerns become more localized, it is easier for a programmer to navigate the system and to evolve a particular functionality in the system without affecting unrelated code bases. Our research will therefore focus on this class of *consistent behavior* concerns. More specifically, we want to assess the refactorability of these crosscutting clusters, and provide the mechanisms for the identification and extraction of these clusters that are most convenient for software evolution and potential aspect refactoring purposes.

We define such crosscutting clusters as *refactorable crosscutting concern* (rCCC):

> A cluster of code that consists of multiple adjacent or proximate method invocations that crosscuts multiple locations in a recurring pattern and that can be refactored into aspects using simple pointcut expressions.

However, our previous case study [38] of crosscutting concerns shows that it can be difficult, if not impossible, to completely encapsulate all crosscutting clusters of a concern into an aspect without creating a complex and rigid pointcut expression.

1.2 An example of refactorable CCC

In the case study we mentioned in the previous section, we studied the nature of cross-cutting concerns in existing systems, and investigated the presence of refactorable cross-cutting concerns in an open-source Java project called FreeMind[5]. From the results of the aspect mining analysis (see Section 2.1 for details), we noticed a group of methods that were consistently executed in the same sequence. We classify these method invocations into the same concern because the invoked methods belong to the same class and two of them have `start-` and `end-` prefixes. The aspect-mining results show that these three methods are called in conjunction across 32 different method declaration bodies(see Table 1.1).

Table 1.1: Summary of the fan-in values of the method calls in the *Action Control* concern of FreeMind

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

Listing 1.3 shows an instance of the concern, where a call to `executeAction()` is preceded by a call to `startTransaction()` and followed by a call to `endTransaction()`. This sequence of invocation is consistently found in 32 instances. Moreover, in most cases, these three methods are called consecutively. Initially, we believed that the simplicity of this concern would make it straight-forward to refactor this concern into aspect, using Java-based aspect extension, such as AspectJ.

[5]See http://freemind.sourceforge.net

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

Listing 1.3: An illustration of Transactional Control concern in `AddArrowLinkAction` class, *FreeMind*

Challenges for AO refactoring The code for this concern exhibits a *consistent behaviour* and the method signatures reveal that the concern serves as a kind of transactional control. Initially, we believed that there are two approaches to extract the crosscutting code into an aspect:

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

Around advice is an type of advice that surrounds a join point such as a method invocation, and performs custom behavior before and after the specified join point. In this case, the join point will be the the location of the methods that contain these two methods. This options requires that `startTransaction()` and `endTransaction()` always be located at the start or end of the method body, which does not hold true for every instance of this concern.

Listing 1.4 shows a variant of the transactional control concern in a class in FreeMind. In this file, the concern code is contained within a `try` block and an `if` statement. For this instance, it is not clear how it can be refactored into as aspect.

```
public Transferable cut(List nodeList) {
  c.sortNodesByDepth(nodeList);
  Transferable totalCopy = c.getModel().copy(nodeList, null);
  try
  {
    /*** initialization code omitted for brevity  ***/

    if (doAction.getCompoundActionOrSelectNodeActionOrCutNodeAction().size() > 0)
    {
      c.getActionFactory().startTransaction(text);
      c.getActionFactory().executeAction(new ActionPair(doAction, undo));
      c.getActionFactory().endTransaction(text);
    }
    return totalCopy;
  } catch (JAXBException e) {
    e.printStackTrace();
  }
  return totalCopy;
}
```

Listing 1.4: An example of a code fragment of the crosscutting concern found in a complex control flow structure in `EdgeColorAction` class, *FreeMind*

Approach 2. Refactor the `startTransaction()` *and* `endTransaction()` *into an 'around' advice of* `executeAction()`.

While this second approach solves the problem in the example of Listing 1.4, it does not solve all types of crosscutting concerns found in FreeMind. For instance, there are alternative implementations of the transactional concern, in which some unrelated statements interleave with the statements belonging to the concern (see Listing 1.5).

The fan-in analysis (see Table 1.1) also shows that `executeAction()` has a higher fan-in value than the other two methods, and we found several instances in the system where `startTransaction()` and `endTransaction()` are not invoked in conjunction with `executeAction()`. A pointcut expression that must cover these exceptional scenarios would be difficult to create.

```
public void setNodeText(MindMapNode selected, String newText)
{
  String oldText = selected.toString();
  try
  {
    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);
    c.getActionFactory().executeAction(new ActionPair(EditAction, undoEditAction));
    c.getActionFactory().endTransaction(c.getText("edit_node"));
  } catch (JAXBException e) {
    e.printStackTrace();
  }
}
```

Listing 1.5: An example of the *transactional concern* of `EditAction` class ,*FreeMind*, where the code that belongs to the concern are not consecutive

Refactoring solution From the above examples, we can generalize that there are two common issues in refactoring crosscutting concern into aspects in a legacy system:

1. The code fragment of the concern is located at an arbitrary location in the declaring method body,

2. The code that forms the concern is not continuous, but interleaved with other code.

Therefore, the first step of refactoring must resolve these problems In most cases all three calls in the cluster are adjacent to each other, and the best approach is to ‘correct’ the type of variant found in Listing 1.5 and change the locations of `startTransaction()` and `endTransaction()` to make them adjacent to the `executeAction()` call. The main challenge of this change is that we need to verify if the statements reordering does not introduce unexpected side-effects. For this particular concern, we manually verified that the reordering was safe.

After the statements are reordered, the sequence of three calls becomes consecutive. However, the goal is to be able to capture the joinpoint of the crosscutting methods using one single pointcut expression. Instead of specifying the name of either three methods in

the pointcut, we opted for another approach: we used the *Extract Method* technique to extract the cluster in each class into a new method called `runTransaction` (see Listing 1.6). After the refactoring, we can create a simple pointcut expression that intercepts all crosscutting clusters (see Listing 1.7).

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

Listing 1.6: The extracted method of the *transctional concern*

```
pointcut actionControl(): call(* *.runTransaction(ActionPair, String, String));
```

Listing 1.7: Pointcut expression that specifies the joinpoint of extracted method in Listing 1.6

The main reasons for extracting the crosscutting method calls into a new methods are:

- The crosscutting code is clustered and its behavior becomes consistent across its callers;
- The intent of crosscutting clusters are disassociated from the original context into a distinct method instance, which reduces the effort for program comprehension in an OOP perspective.
- The addition of the new join point (the newly extracted method) allows programmers to devise a simple pointcut expression that intercepts the crosscutting code. We reduce the needs of using a combination of complicated pointcut expressions, or pointcuts that contain wildcard characters in method name, which may introduce undesirable side-effects such as creating a larger matching join point set than it originally intended [1].

In our study, we concluded that due to the variations in the code, it was difficult to choose a refactoring that neatly encapsulated the crosscutting concerns in a system without sacrificing the readability and simplicity of the pointcut descriptor. To mitigate this challenge, we concluded that it was necessary to use a combination of statement reordering and *Extract Method* refactoring to resolve these variations so that AO refactoring could be applicable.

1.3 Overview of the dissertation

Our research focuses on the refactoring of the third type of *consistent behavior* concern, because we believe that crosscutting code clusters belong to a class of crosscutting concerns that can be located, extracted, and refactored into aspects using simple pointcut expressions. Furthermore, the identification and the transformation process of such concerns can be automated with minimal human guidance and domain knowledge. In this dissertation, we describe a "concern extraction" technique for assessing the refactorability of the crosscutting candidates that are obtained from aspect-mining tools, and applying OO transformation to targeted crosscutting code such that it can be easily refactored into maintainable aspects. In the course of this work, we will address the following three questions:

1. How can we automatically assess the results of an aspect miner and distinguish *refactorable crosscutting concerns* among the candidates?

2. How can we automatically apply OO transformations to discover and resolve the variations in the crosscutting code and isolate its intent by extracting the targeted code cluster into a separate method?

3. Is the concern extraction technique sufficient for reducing the complexity of the potential pointcut expressions and improve the refactorability of the concern?

The goal of our research is to utilize the data from aspect-mining and identify refactorable crosscutting concerns that facilitates the potential refactoring process. We present

ConcernExtractor, a tool that:

1. Automatically identifies the presence of crosscutting code clusters in a system,
2. Assesses if the concerns can be extracted using various metrics, and
3. Performs code transformation that extracts the crosscutting clusters into standalone methods that share a common method name and parameters.

The *ConcernExtractor* is implemented as a plugin for the Eclipse IDE, and uses the existing refactoring modules in the Eclipse framework for code transformation.

We have implemented our technique and applied it to identify refactorable concerns from the results of an aspect-mining tool. We extracted the crosscutting concern candidates from 5 medium-size (20 - 80 KLOC) Java projects using an open-source tool developed by a different team.

Our contributions include a description of our automatic technique for the detection of assessment and code transformation process and a detailed account of the validation procedure of this technique in existing systems.

In Chapter 2, we describe foundational works that motivate and complement our research. In Chapter 3, we provide the details of our technique for identifying and assessing the refactorability of the crosscutting code cluster. We describe a quantitative evaluation of the effectiveness of our technique, and the applicability of ConcernExtractor in Chapter 4, and conclude in Chapter 5.

Chapter 2
Background

Aspect-oriented programming (AOP) is a programming paradigm that enables a modular implementation of crosscutting concerns in a system [21]. *Crosscutting concerns* are functionalities that need to be split up and integrated into different modules . Services such as logging and transaction control are often contained by methods from independent modules. The presence of crosscutting concerns violates the principles of *separation of concerns* and leads to *code scattering* and *tangling* [30]. Crosscutting concerns reduce the readability and maintainability of a system. AOP proposes a solution by encapsulating the crosscutting concerns from a system into new modular units called *aspects* [21]. AspectJ [20] is the most popular AOP extension of the Java languages and it provides new programming constructs such as *pointcut expressions* and *advices* to implement crosscutting code in aspects.

Aspect-oriented (AO) refactoring [22] synergistically combines AOP with object-oriented refactoring practices [13] to extract crosscutting elements from existing systems to improve modularity. The study of AO refactoring can be further categorized into two major branches of research: *aspect mining*, which studies the techniques for locating the crosscutting concerns in existing systems, and *aspect refactoring* or *aspect extraction*, which explores the patterns and practices for separating the crosscutting concern code from the core module into aspects [28](Ch. 9).

2.1 Aspect mining

Aspect mining is the methodology for automatically identifying the crosscutting concerns in source code. Aspect-mining techniques are primarily intended for refactoring crosscutting concerns into aspects. Early aspect-mining tools such as Aspect Mining Tool (AMT) [17] use query-based approaches and locate crosscutting elements based on textual and type similarities in the source code. However, later aspect mining researches focus on a complete automation of the mining process. Marin et al. [27] introduce *fan-in analysis*, which determines the degrees of scattering of code based on the number of times a method is being invoked throughout the project. Since method calls are the simplest program element that can be refactored using pointcut-based AO constructs such as AspectJ, the fan-in values of crosscutting methods provide a quantified assessment of the potential of AO refactoring, and motivates our research to assess the refactorability of tightly coupled methods with high fan-in values. The fan-in analysis is implemented as a part of the FINT framework[1]. The FINT framework is used as the primary aspect miner in our research.

A version-history based approach by Breu and Zimmermann [5] analyzes the addition and evolution of program elements over time and correlates this data with the author and timestamp data from the version history. This approach is more scalable in large projects and its precision in locating crosscutting concerns increases with the project size and history. Zhang et al. [40] use a *random walk* algorithm to explore the incoming (*popularity*) and outgoing (*significance*) of references to each program element. Their random walk algorithm differs from other syntax-based approaches mentioned above because it attempts to distinguish crosscutting elements from those represented in the core functionality of the system. *Timna*, by Shephard et al. [34], is a framework that uses machine learning techniques to augment the precision of other aspect mining techniques such as fan-in analysis. The drawback is that it requires a user to manually tag relevant program elements for the offline training phase.

Many aspect-mining research project apply *formal concept analysis* (FCA) [14] to explore relationship between different program elements and identify aspects. Tonella

[1] http://swerl.tudelft.nl/bin/view/AMR/FINT

and Ceccato [37] use concept analysis to analyze the execution traces from use cases and identify crosscutting elements. *DynAMit*, created by Breu et al. [4], is another FCA tool based on trace history. However, in general the runtime of FCA is exponential [23] and therefore its applicability is constrained to analyses that produce a small dataset, such as dynamic analysis. The FINT framework also includes a concept analysis tool [26] that locates the a group of crosscutting method calls in a system based on their fan-in values. We use FINT to extract the crosscutting candidates because it generates a comprehensive set of method calls that are both crosscutting and highly coupled. However, the exponential complexity of FCA places a constraint on its applicability in large systems.

2.2 Refactoring and AOP

Identified crosscutting concerns can be refactoring into aspect either manually or with automated refactoring tool. Monteiro et al. [30] present a collection of low-level AO refactoring techniques that solve specific crosscutting symptoms in systems. At the design level, Hannemann et al. [18] solve the code scattering and tangling that are introduced through design patterns by reimplementing the patterns using AspectJ. Several case studies [3, 24, 29] describe the application of these AO refactorings in a small system to demonstrate how AO refactoring can remodularize crosscutting concerns. Using techniques described in [30], Monteiro et al. [29] illustrate the refactoring process in a simple example that implements the Observer pattern. Marin [24] and Binkley et al. [2] refactor the Undo concern in JHotDraw manually and with tool-support, respectively. However, all authors conclude that the refactored aspect code will need additional refactoring to simplify the pointcut expression and the architecture of the aspect classes.

AO refactoring in large systems is more complex because of the variations in the crosscutting concern implementation. Bruntink et al. [6] investigate the tracing concern in C-based components and show that it exhibits significant variability which makes it difficult to refactor into aspects. Colyer et al. [9] manually refactor the Enterprise JavaBean^{TM} support component in a large Java middleware system using AspectJ. However, the AO

refactoring involves a heavy use of intertype declaration compared to the use of advice. They argue that the design and flexibility of AO refactoring will be superior to traditional object-oriented refactoring. Both studies rely on the authors' knowledge of a particular non-invasive concern and the experience is not applicable to more domain-specific crosscutting concerns.

2.3 Tool-based AOP refactoring

Automation reduces the efforts in refactoring crosscutting concerns into aspects. However, human guidance in the process is usually necessary: a developer needs to verify the identified crosscutting concerns and evaluate if refactoring is desirable. AOPMigrator [3] is a semi-automated Eclipse-based refactoring toolkit that implements low-level code transformations and automatically infers the AspectJ implementation of the annotated Java code fragments. Nevertheless, each refactoring is applicable to only one instance of code fragment and cannot extract multiple crosscutting instances. The generated aspects are often not intuitive and require manual fine-tuning. Hannemann et al. implement role-based refactoring [19] to address the crosscutting problem from a modularization perspective. A user needs to map selected program elements according to a set of pre-defined schema, and the tool automatically generates the aspect implementation of the mapping and performs the code transformation based on the user input. This technique is most appropriate if the predefined schema match exactly with the mapped elements, such as canonical implementations of design patterns, but its applicability in the variant implementations remains an open question. The authors [3, 16] all conclude that OO transformations should be applied extensively to reduce the code complexity such that the AO refactoring actually simplifies the design. Our research is motivated by the benefit in extracting highly tangled method calls using OO refactoring to reduce the complexity in subsequent AO refactoring.

Another major hurdle in AO refactoring is in devising pointcut expression that can describe the targeted joinpoints and is comprehensible and maintainable. Anbalagan et al. [1] propose an automated approach that uses the results from an aspect mining tool and

infers pointcut expressions for these aspects. The inference engine performs a clustering phase based on the textual and syntactic similarity of the pointcut, and outputs the most succinct form that crosscuts all targeted joinpoints. However, the authors show that the average size of the pointcut statement remains quite complex (> 8 pointcut expressions) in large projects despite the clustering. This work inspires us to consider approaches that can unify different crosscutting methods into one method signature, which simplifies the pointcut expression that will be needed to refactor the crosscutting code into aspects.

Chapter 3
Concern Extraction Techniques

Aspect-mining techniques generally identify crosscutting concerns based on the *scattering* [8, 27] or the *coupling* [4, 5] of different method calls. However, our main focus is the presence of clusters with multiple method invocation statements in the code base. Moreover, these methods should be invoked sequentially or consecutively in a recurring pattern. We set the refactoring targets on multiple method invocation statements because they imply a large functionality that requires multiple steps. More importantly, if the method calls in the clusters are invoked consecutively, then these statements would be extractable into a separate instance such as a method call or an advice in an aspect.

The goal of our work is to be able to identify this type of *refactorable* crosscutting concern in a system and extract these crosscutting clusters across different locations into standalone methods, so that these concerns are relatively isolated from the core function, are easily identifiable, and can be readily refactored into aspects if necessary.

For a candidate concern containing multiple method calls that are considered to be crosscutting either manually or automatically by aspect-mining tools, the concern extraction tool should:

- Identify the presence of refactorable crosscutting cluster in a code base and provide visualization and navigation support of the targeted cluster code.
- Automatically assess if the concern could be extracted based on various metrics.

- Perform the code transformation that extracts the crosscutting clusters into standalone methods with a unified method name and parameters.

The following section describes the details of our approach, implemented in the *ConcernExtractor*, a plugin for the Eclipse IDE that provides assessment and extraction mechanisms for crosscutting concerns. The ConcernExtractor is built upon the *ConcernMapper*, a bookmarking tool for programmers to organize program elements (i.e. class, method, or field) that are considered related into arbitrary modules called *concerns* inside the Eclipse IDE (see Section 3.2 for a detailed description of the ConcernMapper).

The rationale behind building our technique in the Eclipse framework is that the Java Development Tool (JDT) package of Eclipse provides powerful source code search and manipulation functionalities that we can leverage. Furthermore, various aspect-mining tools [5, 27] are already built on top of Eclipse. Therefore, with minor modifications, results from the aspect-mining tools can be exported into an XML format that is readable by the ConcernMapper model. The ConcernMapper can quickly group elements that are considered crosscutting into a "concern" node in the model. The ConcernExtractor then analyzes the elements in each concern and assesses the possibility of extracting them into aspects.

3.1 Crosscuting concern assessment and extraction

The assessment and extraction technique in ConcernExtractor is divided into a four-stage process:

1. Identify crosscutting method calls in the source code.
2. Analyze and match code with similar structures into crosscutting clusters of code.
3. Perform flow-analysis on non-consecutive code snippets to assess if the snippet can be *clustered* and extracted into a method.
4. Extract the selected code snippets into a new method.

```
void run(Foo f)
{
  f.bar(); // a method cluster
  f.foobar();
}
```

Listing 3.1: A method cluster with consecutive seed methods

```
void run2(Foo f)
{
  f.bar(); // a method cluster
  System.out.println(''Hello World'');
  f.foobar();
}
```

Listing 3.2: A method cluster with non-consecutive seed methods

Before we elaborate on the detailed implementation of each stage, we define a number of key terms:

Definition 1: Seed method A seed method is a method that is called multiple times in different method declaration bodies and is identified as crosscutting.

Definition 2: Method cluster A method cluster consists of multiple method invocation statements that invoke the seed methods, and are located within the same block The reason for considering only method invocation statements within the same block is that it is the simplest form of multiple statement execution that we can consider as "crosscutting".

For example, assume that there are two seed methods, namely `bar()` and `foobar()`. In the declaration of method `run()` (see Listing 3.1), since both method statements of `bar()` and `foobar()` are invoked consecutively, these two statements constitute a *method cluster*.

Two seed method statements may form a method cluster even if they are not consecutive, as long as they are syntactically located in the same block.

However, the method statements of `bar()` and `foobar()` in Listing 3.3 are not considered a method cluster, because the statements are not located inside the same block.

```
void run3(Foo f)
{
  f.bar(); // a method cluster
  if (f.toString() != null)
  {
    f.foobar();
  }
}
```

Listing 3.3: A method that contains seed method statements, but does not contain a method cluster

Definition 3: Statement reaggregation *Reggregation* describes the code transformation process that rearranges the order of statements inside a Java block, such that certain targeted statements become adjacent to each other. However, the invocation sequence of the target statements should be preserved.

Figure 3.1 shows an example of statement reaggregation, in method `run()`, where the statements in the method body of `run()` need to be swapped such that the statement that contains `Foo.bar()` is adjacent to the method statement of `Foo.foobar()`. After the *reaggregation*, the statements in the body are swapped, but the call to `Foo.bar()` still precedes to the call to `Foo.foobar()` — the sequence of the execution of these two methods remains the same after the reaggregation.

```
void run(Foo f)
{
  f.bar();
  // irrelevant code
  f.foobar();
}
```

⇒

```
void run(Foo f)
{
  // irrelevant code
  f.bar();
  f.foobar();
}
```

Figure 3.1: Effects of statement aggregation

Definition 4: Isomorphic clusters Two method clusters are *isomorphic* if

1. they contain the same number of statements,
2. The method invocations statements are invoked in the same sequence for both clusters, and
3. For each method invocation statement in a cluster, there is a corresponding statement in the other cluster that has the same method signature in its method invocation expression.

Listings 3.4 and 3.5 show two instances of method declaration bodies in *Freemind*. In the code snippets, the method calls that are underlined represent the seed methods. In both scenarios, each declaration contains a method cluster that consists of `startTransaction`, `executeAction`, and `endTransaction` statements. Also, these three methods are invoked in this order respectively. In this example, the clusters in both snippets are isomorphic, since the method signatures of the seed methods in both clusters are isomorphic. For instance, the `executeAction` statements in both clusters are isomorphic because the identifiers of the `executeAction` method in both clusters belong to the `MindMapNode` type, and each parameter of the method expects expression that returns an `ActionPair` object (see Section 3.3 for more details).

```
1
2  public void setNodeText(MindMapNode selected, String newText)
3  {
4    try
5    {
6      c.getActionFactory().startTransaction(c.getText("edit_node"));
7      EditNodeAction editAction = c.getActionXmlFactory().createEditNodeAction();
8      editAction.setNode(c.getNodeID(selected));
9      editAction.setText(newText);
10
11     EditNodeAction undoEditAction = c.getActionXmlFactory().createEditNodeAction();
12     undoEditAction.setNode(c.getNodeID(selected));
13     undoEditAction.setText(oldText);
14
15     c.getActionFactory().executeAction(new ActionPair(editAction, undoEditAction));
16     c.getActionFactory().endTransaction(c.getText("edit_node"));
17   } catch (JAXBException e) {
18     e.printStackTrace();
19   }
20 }
```

Listing 3.4: An instance of crosscutting method cluster in `EditAction` class, *FreeMind*

```
1
2  public void addLink(MindMapNode source, MindMapNode target)
3  {
4    String value = (String) getValue(NAME);
5    ActionPair actionPair = getActionPair(source,target);
6    String value_2 = (String) getValue(NAME);
7
8    modeController.getActionFactory().startTransaction(value);
9    modeController.getActionFactory().executeAction(actionPair);
10   modeController.getActionFactory().endTransaction(value_2);
11 }
```

Listing 3.5: An instance of `executeAction` method statement in `AddArrowLinkAction` class, *FreeMind*

3.2 Analyzing crosscutting candidates

To better qualify the identification of *refactorable crosscutting concern*, we define the following criteria necessary for a concern candidate to be crosscutting and contain crosscutting method clusters:

1. It consists of at least two distinct method calls in separate code statements.

2. The statements are always executed in the same sequence.

3. The target statements are executed consecutively (one call is immediately followed by another) at least at one location in the project. The presence of consecutive statements suggests a behavior that is composed of multiple actions/method calls. If this consecutive pattern is found in at least one location, we can be more confident that these seed methods actually form a concern.

4. The target clusters or sequences are found in at least three distinct locations[1]. Although a cluster can be described as "crosscutting" if it is found in more than one instance in the source code, we believe that the presence of three distinct instances makes a stronger case for a crosscutting concern.

Initially, we wanted to create a user-friendly approach to let programmer declare a set of program elements as crosscutting candidates for extraction. This requirement motivated us to built the ConcernExtractor as an extension of ConcernMapper [33] (see Figure 3.2 for a screenshot of ConcernMapper). ConcernMapper provides a lightweight approach for concern modeling by enabling programmers to drag-and-drop program elements (fields or methods) in a project into a separate view for quick referencing and navigation inside the Eclipse IDE. Program elements that are considered as related can be dragged into an node called "concern". The primary objective of the ConcernMapper is to let programmers record and keep track of program elements from different classes or files into one view to reduce the efforts in source comprehension and navigation.

Nevertheless, some aspect-mining techniques, such as dynamic traces [4, 37], do not provide any additional information other than suggesting some elements are related. A "concern", therefore, serves as the starting point for grouping seed methods. The responsibility of the ConcernExtractor is to locate the call sites of the seed methods to check if any forms a method cluster.

ConcernExtractor uses the Java search engine in the Eclipse JDT component for searching the locations in the project that reference the seed methods. The Java search engine

[1]Our initial investigation showed that the number of refactoring crosscutting concern decreases sharply if the criterion is higher than three.

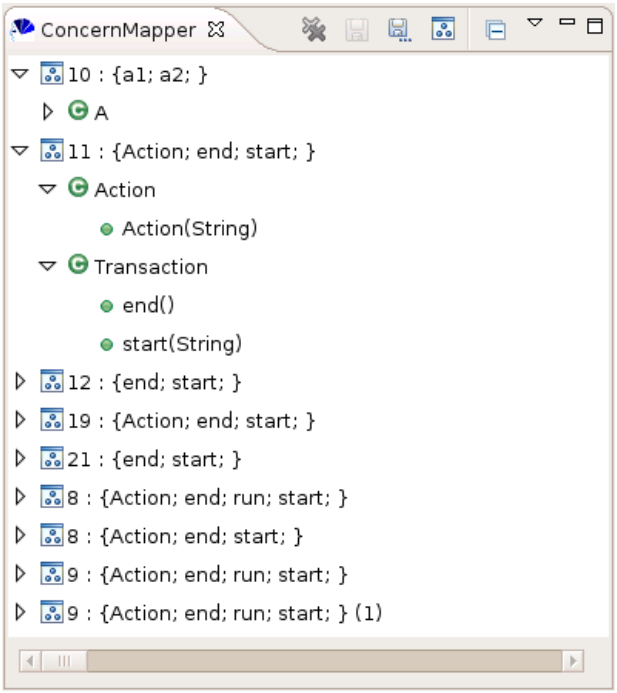

Figure 3.2: ConcernMapper

returns the method declarations and their corresponding source files that contain calls to seed methods. ConcernExtractor then analyzes these source files and tries to search for method clusters from the method declaration bodies that call the seed methods.

This information provides the minimally-required scope to construct method clusters in the source files. In the ConcernExtractor, two statements constitute a *method cluster* if:

1. Both statements contain calls to the seed methods,
2. Their innermost containing blocks are the same, and
3. None of these statements contain an inner block.

   (According to the *Java Language Specification* [15], a *block* is a sequence of statements, local class declarations and local variable declaration statements within braces.

Examples of statements that contain an inner-block include anonymous class, `if` and `while` statements and `for` loop.)

By confining the scope to statements within the same block, we can start exploring the possibility of refactoring these statements into a new method (discussed in Section 3.5). This type of refactoring is called *Extract Method* [13], and the purpose of this refactoring is to turn a snippet of code into a standalone method that can better explain its purpose. In the context of our research, the purpose for the refactoring is to extract crosscutting clusters into methods that share common name that can better express the intent the crosscutting concern.

However, it is not always possible to extract any arbitrary code snippets into new methods: a snippet contained in a complex control flow structures, or snippets that have multiple return statements, will be not be considered extractable by the JDT Extract Method refactoring. To make sure that every instance of the *method cluster* is extractable, we only consider isomorphic clusters that only contain method invocation statements. From a CCC perspective, a sequential execution of multiple statements that is consistently being invoked across different modules is the most common form of crosscutting behavior that sufficiently justifies the method extraction.

However, our criteria allows a method cluster to contain statements that are not consecutive. In this case, a cluster may not be immediately extractable. In Section 3.4, we will discuss how to resolve the interleaving between these statements through *statement reaggregation*.

3.3 Matching isomorphic clusters

After obtaining the method clusters, the next stage in the concern extraction involves assessing if any two clusters have the same behavior and belong to the same crosscutting concern. To verify if the clusters have the same behavior, we need to check that the statements in the clusters have the same method signature (see the definition of *isomorphic clusters*) which is done through examining the abstract syntax tree (AST) of the statements

in both clusters to see if the structure of the statements are equivalent in both clusters, and therefore produce similar behaviors.

The Eclipse SDK provides an `ASTMatcher` class to compare if two ASTs are *structurally isomorphic*, meaning that the AST structures are identical. However, requiring two AST to be structurally isomorphic is too constraining for our purpose. To illustrate, we will examine the issue with the isomorphic clusters more deeply in Listings 3.4 and 3.5.

Figures 3.3 and 3.4 show the AST structures of the `executeAction` statements in both instances. The AST nodes from both statements are identical at the root and at the identifier level (a.k.a. left sub-tree). However, the differences between both statements are at the parameter of the `executeAction`.

Although both statements accept one `ActionPair` object, line 15 of Listing 3.4 shows that the parameter of `executeAction` is not an identifier, but a constructor call to an `ActionPair` object, which itself contains two arguments of `EditNodeAction` type. However, both statements invoke the same function and should be considered as belonging to the same concern. Therefore, it requires a more liberal matching scheme such that these two statements can be considered *isomorphic* in our model. For our assessment purpose, we relax the requirement of *isomorphic* AST to only consider the root node of the AST and the datatype of the subnodes. For instance, the *executeAction* statements are considered isomorphic because the identifiers of the method statement belong to `MindMapNode` type, and the expressions in the arguments return an `ActionPair` object. Therefore, the customized AST matchers only verifies that both method invocation statements have the same method signature. This relaxed isomorphic requirement is adequate to ensure the clusters that are isomorphic will have the same behavior, and that these clusters can be made *structurally isomorphic* (identical AST structure in the clusters) through code transformation, such as by extracting all the identifier and argument expressions the statements into local variables.

Our customized AST matcher initially selects a cluster that contains only consecutive statements as the match target, such that it guarantees at least one crosscutting cluster is extractable (since *Extract Method* refactoring only targets a code snippet that contains

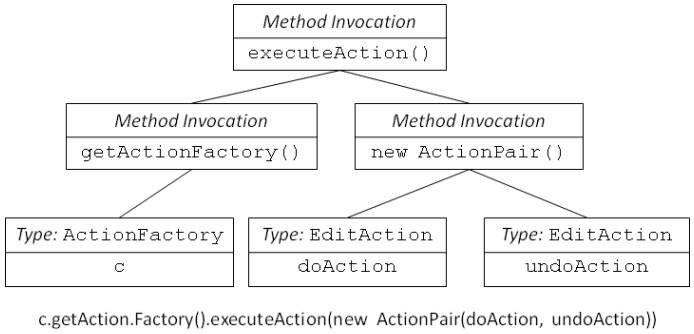

Figure 3.3: AST structure of the `executeAction` statement in Listing 3.4

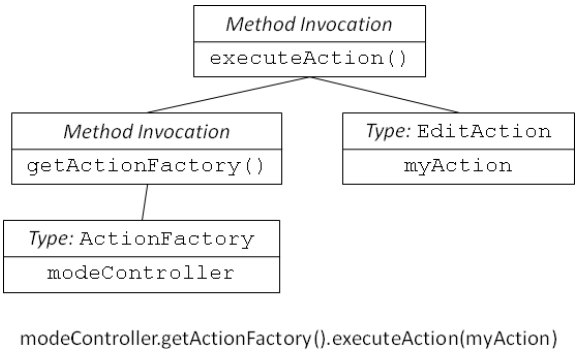

Figure 3.4: AST structure of the `executeAction` statement in Listing 3.5

consecutive statements). It then performs a statement-by-statement match with all other method clusters and checks if any cluster contains a subset of statements that is isomorphic to the original cluster. An isomorphic cluster should produce the same behavior as the target cluster if the seed method statements are executed successively and therefore they belong to the same crosscutting concern.

3.4 Statement reaggregation

Before the clusters are extracted, we need to perform an intermediate code transformation to ensure that the statements in the clusters are consecutive. The *Extract Method* refactoring turns a continuous fragment of code into a new method. However, not all statements in the same isomorphic cluster set are necessarily contiguous. The statements of a target cluster may be interleaved with unrelated statements, such as logging calls or local variable declaration code. (In fact, it is often a justified coding practice to declare local variables just before they are referenced to reduce their *span* [10].)

Statement reaggregation involve rearranging the execution order of the statements in the enclosing block of the target cluster, such that all of its non-consecutive statements are moved to a location adjacent to the last target statement in the cluster. Listing 3.6 shows the effect of reaggregation on the method cluster in the `EditAction` class found in Listing 3.4. After reaggregation, all of the seed method statements in the cluster become consecutive in the declaring method body.

```
public void setNodeText(MindMapNode selected, String newText) {

  String oldText = selected.toString();
  try
  {
    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);

    // Reaggregated snippet
    c.getActionFactory().startTransaction(c.getText("edit_node"));
    c.getActionFactory().executeAction(new ActionPair(editAction, undoEditAction));
    c.getActionFactory().endTransaction(c.getText("edit_node"));
  }
  catch (JAXBException e) {
    e.printStackTrace();
  }
}
```

Listing 3.6: Reaggregated snippet of the example in Listing 3.4

In general, we consider it "safe" to push a particular statement in a method body further in the execution order (i.e. cut a particular line, and paste it somewhere after its previous position in the same declaration), only if it is syntactically correct, and satisfies either of these conditions:

1. It does not write to any variable that are referenced after its previous position , or;
2. The variables that it writes to are not read by any statements that precede its new position.

If either condition is not met, then it is almost certain that for the same input, the output of the function and the internal state of the system will be different from the unmodified version of the method. Since the order of which the variables are read or written is disrupted, the previous assumptions of the logic of control flow could become invalid, and we cannot assume that the modified code replicates the same behavior as the unmodified version. To prevent the unintended change in behavior due to reaggregation, our extraction technique would perform a flow analysis on the clusters that contain isomorphic but non-consecutive statements, such that reaggregation is only "safe" if the targeted statement($stmt_i$) and all the interleaving statements (int_i) between the target statements and the final statement of this isomorphic set ($stmt_f$) do not *interfere*:

$$readSet(stmt_i) \neq writeSet(int_i)$$
$$writeSet(stmt_i) \neq readSet(int_i)$$
$$writeSet(stmt_i) \neq writeSet(int_i)$$

Nevertheless, even if the flow analysis confirms there is no interference, it does not guarantee the reaggregation is truly safe: the targeted statements maybe contain subtle side-effects that change the states of some other variables not in the scope of the cluster, and rearranging the execution order of these statement maybe create unintended behavior. However, a full-scale side-effect analysis on the affected statement incurs excessive runtime penalty and produces an over-conservative result, which contradicts the spirit of our

lightweight approach. Therefore, we compromise for this deficiency by explicitly prompting a user for enabling reaggregation for each non-consecutive target cluster deemed reaggregable by flow-analysis. It is up to the programmer to decide if reaggregation is suitable in the context of each non-consecutive cluster.

3.5 Extracting isomorphic code snippets

Reaggregation solves the interleaving problem in non-consecutive clusters and all isomorphic clusters that are subject to extraction should now only only contain consecutive statements. For the extraction phase, we reuse the Eclipse JDT refactoring components that are available for extracting the clusters, rather than to rewrite our own refactorings.

However, the Extract Method refactoring implementation in JDT is insufficient because it often does not recognize two code fragments as duplicate when the behavior of the two fragments are essentially equivalent. For instance, although the snippets in Listings 3.4 and 3.5 are essentially equivalent in behavior, the extracted forms of both snippets would not be identical (see Listings 3.7 and 3.8 for the extracted form of the method by directly applying the JDT *Extract Method* refactoring to the snippets). Moreover, if *Extract Method* refactoring is directly applied to the method clusters, we cannot guarantee that the signature of the extracted methods are all identical. This outcome is undesirable for aspect refactoring purpose, because it is not always possible to create a single pointcut expression that intercepts the extracted methods and capture **all** of their arguments. Without knowing all the arguments that are passed to the extracted method, it is impossible to refactor the code from the extracted methods to an advice.

```
public void runTransaction(Action doAction, Action undoAction)
{
  c.getActionFactory().startTransaction(c.getText("Edit_note");
  c.getActionFactory().executeAction(new ActionPair(doAction, undoAction));
  c.getActionFactory().endTransaction(c.getText("Edit_note");}
}
```

Listing 3.7: The extracted form of the method cluster in Listing 3.6

```
public void runTransaction(string value1, Action myAction, string value2)
{
  modeController.getActionFactory().startTransaction(value1);
  modeController.getActionFactory().executeAction(myAction);
  modeController.getActionFactory().endTransaction(value2);}
}
```

Listing 3.8: The extracted form of the method cluster in Listing 3.5

O'Connor et al. [31] investigated the internal implementation of JDT and identified cases when the JDT Extract Method refactoring will not recognize potentially duplicate code in the same source file. One of the findings in the study is that the JDT Extract Method refactoring often does not consider two subnodes in two different ASTs that are of the same scope and the same type identical. The problem is that an AST node that belongs to the `SimpleName` type, (which represents a local variable or a field in the JDT terminology), only matches with another `SimpleType` instance, but not with an expression that returns the same datatype. This explains why JDT refactoring does not match the otherwise identical code snippets in 3.7 and 3.8. To solve this inadequacy, the paper describes a sequence of micro-refactorings that converts the selected code into a normalized form, such that the JDT Extract Method refactoring algorithm will considered two matching snippets as equivalent.

Inspired by their approach, ConcernExtractor implements a similar refactoring strategy that normalizes the targeted crosscutting snippets. A normalized statement in a method cluster contains one method invocation expression with an identifier as the instance, and local variable(s) as its argument(s). Through normalization, each statement in a cluster becomes *structurally isomorphic* to its counterpart in the other cluster. The JDT Extract Method algorithm will recognize that the isomorphic snippets as duplicate, and the extracted forms of these snippets will have a common method name and parameters.

3.5.1 Extracting instance and arguments into locals

Recall from the previous section that our isomorphic AST matcher only checks the data-type of the instance and arguments of a seed method, regardless of whether it is an expression or a variable. In order to normalize the statement, ConcernExtractor needs to convert the AST nodes that represent the instance and arguments of the seed method into local variable using the *Extract Local Variable* refactoring. The *Extract Local Variable*[2] refactoring takes an expression that is being used directly and assigns it to a local variable first and this variable is then used where the expression used to be, and is also implemented in the Eclipse JDT package. Local variable extraction does not change the semantics of the targeted clusters when it is performed on expressions that only need to be evaluated once in the block that contains the cluster. After the extraction, all the nodes that represent the instances and the arguments of the seed methods will be of type `SimpleType`. The JDT Extract Method refactoring can then recognize the targeted crosscutting clusters that are in the same source file as duplicates. Also, the extracted forms of the crosscutting clusters are guaranteed to have the same parameters.

3.5.2 Rearranging the extracted local declarations

One side effect of *Extract Local Variable* refactoring is that the local variables are declared just above the statement where it get extracted from. Listing 3.9 shows the result of the *Extract Local Variable* refactoring when it is applied automatically by the IDE to the arguments of the seed methods in `EditAction.setNodeText`. After the refactoring, the seed methods invocation are interleaved with the local variable declarations, and the statements of the method cluster are not invoked consecutively anymore. Since the statements are not clustered, they cannot be extracted in a new method as one snippet.

To deal with this problem, O'Connor et al. create the Code Motion Refactoring, which

[2] See http://www.ibm.com/developerworks/library/os-ecref/.

```
public void setNodeText(MindMapNode selected, String newText)
{
  String oldText = selected.toString();
  try
  {
    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);
    String tmp = c.getText("edit_node");
    ActionFactory factory = c.getActionFactory();
    factory.startTransaction(tmp);
    ActionPair action = new ActionPair(EditAction, undoEditAction);
    ActionFactory factory_2 = c.getActionFactory();
    factory_2.getActionFactory().executeAction(action);
    String tmp_2 = c.getText("edit_node");
    ActionFactory factory_3 = c.getActionFactory();
    factory_3.getActionFactory().endTransaction(tmp_2);
  }
  catch (JAXBException e) {
    e.printStackTrace();
  }
}
```

Listing 3.9: Applying Extract Local Variables on the arguments of the seed methods.

moves a specified snippet of code to a different location. However, it is conceptually equivalent to the Statement Reaggregation in ConcernExtractor, since both transformations attempt to rearrange the execution order of the statements. In this case, we are concerned only with the movement of variable declaration statements; specifically, those statements just created by the JDT Extract Local Variable refactoring. Our aim is to move the local declaration statements directly above the statements to be extracted, so ensure that the extraction target again forms a cluster of consecutive statements (see Listing 3.10). Since the local variables are newly introduced by the Extract Local Refactoring in the previous stage, it is not necessary to use flow analysis to verify the correctness of the reaggregation.

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

    String tmp = c.getText("edit_node");
    ActionFactory factory = c.getActionFactory();
    ActionPair action = new ActionPair(EditAction, undoEditAction);
    ActionFactory factory_2 = c.getActionFactory();
    String tmp_2 = c.getText("edit_node");
    ActionFactory factory_3 = c.getActionFactory();

    factory.startTransaction(tmp);
    factory_2.getActionFactory().executeAction(action);
    factory_3.getActionFactory().endTransaction(tmp_2);
  } catch (JAXBException e) {
    e.printStackTrace();
  }
}
```

Listing 3.10: Reaggregating the temporary local declarations.

3.5.3 Extracting isomorphic crosscutting clusters

After the Statement Reaggregation is performed, the JDT Extract Method refactoring API is called on each identified isomorphic cluster. A user is prompted once to insert the name of the extracted method. After inserting a name, the Extract Method refactoring is automatically applied to all of the targeted clusters with the same method name. The Extract Method refactoring API also handles all the precondition checks that ensure the syntactic correctness of the *Method Extract* process. At the end of the this stage, all isomorphic clusters that were subjected for extraction are extracted into standalone methods that share a common method name and parameters.

Chapter 4
Quantitative Evaluation

Our concern extraction technique is intended to complement aspect-mining technology by identifying crosscutting candidates that are refactorable. Furthermore, it mitigates the difficulties of refactoring to aspects by extracting these crosscutting clusters into methods with a common method signature. It enables aspect-refactoring tools to produce intuitive and understandable pointcuts for the aspects. However, to support our claims, we evaluate our concern extraction approach to answer the following questions.

- *Does concern extraction help distinguish refactorable crosscutting concerns from the results aspect-mining technique?* To answer this question, we apply aspect-mining to a software system and collect program elements that are considered crosscutting by the aspect-mining tool, and applying the ConcernExtractor to this data to obtain the number of refactorable crosscutting method clusters. Using the results, we can evaluate the percentage of the concerns that are considered refactorable over of the mined results.

- *Is flow analysis adequate for determining if non-consecutive crosscutting code is extractable?* For this purpose we collect the number of non-consecutive clusters from the identified crosscutting concerns, and apply flow analysis to these clusters to determine if they can be reaggregated. We then assess the correctness of flow analysis

by manually checking if the statement reaggregation would produce undesired side effects.

- *Is the concern extraction necessary for localizing crosscutting clusters such that they can be refactored to an aspect, or does direct aspect refactoring suffice in most scenario?* In this section we manually check the position of the each crosscutting cluster in its containing block, and evaluate the improvement of the number of clusters that become refactorable to aspect due to cluster extraction, over the clusters that can be directly refactored by simple pointcut expression.

4.1 Experimental Environment

Our evaluation is preceded by an aspect-mining phase which collects crosscutting program elements to serve as the candidate for concern extraction.

We selected FINT[1] as the aspect-mining engine to collect to the initial data for our evaluation, since it only requires the source code of target systems as inputs and is most suitable for our needs. FINT is an aspect-mining research tool that provides a common framework for mining crosscutting concerns using various different techniques from the Eclipse IDE environment [26]. In particular, FINT contains two aspect mining techniques that are useful for our evaluation. FINT implements the *fan-in* analysis (see Section 2.1), which collects and analyzes the number of different method invocations in the system. Furthermore, the *grouped calls* analysis extends the basic *fan-in* analysis, by collecting the *fan-in* of a *group* of method calls — it applies formal concept analysis [23] to a target systems and finds groups of callees (*attributes*) that are consistently invoked together by the same callers (*objects*). Therefore, it requires at least two method calls that are consistently called together to form a crosscutting concern. Compare to the normal *fan-in* analysis, which only counts the number of individual method call sites, the results of grouped calls analysis is more refined.

[1] See http://swerl.tudelft.nl/bin/view/AMR/FINT.

The running time of concept analysis is $O(n^2)$, n being the total number of method invocations in the system. In order to collect adequate data within a reasonable time constraint (within 24 hrs in our evaluation), we limit the size of target systems to 60∼85 KLOC. We used a Linux desktop workstation with a Pentium 4 - 3.40 GHz processor and 3 GB memory as the default configuration of evaluation.

In our initial evaluation we selected 5 open sources projects as the target systems. Table 4.1 contains an overview of the five target systems that we used for the evaluation.

Table 4.1: List of target systems

System	Version	kLOC	Total no. of grouped calls
JBossAOP	4.0	66	924
jEdit	4.3	63	1404
FreeMind	0.8.0	65	241
Ant	1.7	86	1882
ArtOfIllusion	2.4.1	79	3718

4.2 Evaluation procedure and variables

In our evaluation, we only consider groups with a common fan-in of at least 10: a group must contains at least *two* seed methods that share a caller at minimum *ten* different locations in the system. In our evaluation, we turned off the data filter in FINT, which would otherwise ignore the following groups of candidates:

- Candidates that contain methods of the JUnit library (a.k.a. JUnit testcases);
- Candidates that contain Java collection utilities such as `size()`, `iterator()`;
- The callees in a candidate group are always called in a single Java statement.
- Candidates that **only** contain callees with the same prefixes, such as `get-`, `read-` or `write-`.

We disabled the filter to collect as many candidates as possible, and for our evalution, use this value as the total number of grouped calls/concern candidates in the system. However, for practical purpose, any candidates that belong to the groups mentioned above are not considered for extraction. We developed a customized version of FINT that exports the results from the grouped calls analysis to ConcernMapper. Each group is converted into a "concern node" (see Figure 3.2), and its seed methods become the concern elements.

During the evaluation, we evaluated each concern candidate from the ConcernMapper using the ConcernExtractor and obtained the distribution of the crosscutting clusters. Since our objective is to find "crosscutting" methods, and our understanding of "crosscutting" implies that there should be more than two instances of the crosscutting clusters in the system, we only considered concerns that contain at least 3 isomorphic clusters as a valid result. We also did not considers clusters that contain only assignment statements, since they will not be extractable to a method.

4.3 Question 1: Identifying refactorable crosscutting code

Criterion: Effectiveness of ConcernExtractor to identify refactorable crosscutting code.

Using the data collected from grouped calls analysis, we applied ConcernExtractor to compute the number of refactorable clusters. We break down the distribution of the refactorable concerns in evaluation into the following categories in Table 4.2:

Total no. of grouped calls from FINT: the number describes the number of grouped calls obtained from FINT, where each group consists of multiple common methods that are invoked together in the same method body more than once.

Cluster groups with $\geq$ 3 statements: the number of different grouped calls that is considered refactorable by the ConcernExtractor. Each group must consist of at least three methods.

Cluster groups with 2 statements: the number of different grouped calls that is considered refactorable by the ConcernExtractor. Each group must consist of only two methods. Cluster groups with more than three methods are excluded from this set.

$\sum$ **Refactorable cluster groups**: the sum of refactorable clusters.

% of refactable cluster groups: $\frac{\sum Refactorable\ cluster\ groups}{Number\ of\ grouped\ calls\ from\ FINT}$

Table 4.2: Summary of crosscutting cluster distribution in the target systems

Target System	**Size of system** kLOC	**Total no. of grouped calls from FINT**	**Cluster groups with ≥3 statements**	**Cluster groups with 2 statements**	**∑ refactorable cluster groups**	**% refactorable cluster groups**
jEdit	63	1404	2	2	4	0.2%
FreeMind	65	241	2	1	3	1.2%
JBossAOP	66	924	2	3	5	0.5%
ArtOfIllusion	79	3718	2	0	2	0.02 %
Ant	86	1882	2	8	10	0.5%
Average	72	1633	2	3	5	0.3%

We made several observations from the result of the Table 4.2. The first observation is that the number of refactorable cluster groups are scarce in each system, compared with the quantity of concern candidates in each system. This result raises some doubts about the validity and usefulness of our extraction and refactoring technique. First of all, it suggests that the existence of refactorable crosscutting concerns are not prevalent in the target systems, which puts the necessity of the crosscutting concern extraction into doubt. Secondly, the result can alternatively suggest that our criterion for a crosscutting cluster is too conservative, which may miss some opportunities for refactoring (i.e. blocks of code that are repeatedly called).

Nevertheless, we emphasize that our goal is to discover *refactorable* concerns, and that multiple sequential method invocations within a block constitutes the simplest form of code snippet that we consider as extractable. If this criteria is weakened, then even if we can identify code snippets that belong to the same crosscutting concern, it will be difficult to verify if they can always be extracted to a standalone method.

The results collected from grouped calls also convince us that the original aspect-mining results are full of noise. For instance, the aspect-mining result of GUI-based application would contain a large amount of grouped calls that belong to the Java Swing library. Moreover, the FINT's grouped calls analysis uses a coarse criteria to relate two different method invocations: two method invocations are considered related as long as they are called inside the same method definition, regardless of the sequence or the context of the calls. Therefore, systems that contain large methods (which is considered a bad coding practice) will produce a larger quantity of grouped calls than systems that are well refactored and structured. The fact that all of our target systems (which have similar sizes) contain a consistent number of refactorable cluster groups implies the existence of such crosscutting concern in a system with reasonable size. The ConcernExtractor allows a user to reduce the original aspect-mining data to a set of methods that are closely related due to the proximity of their locations.

As another interesting observation, the number of actual refactorable concerns in each system is not proportional to the number of grouped calls. Instead, the value seems to correlate to the size of the system: the percentage of refactorable cluster groups remains consistent for the five target systems, which have similar code size.

4.4 Question 2: Flow Analysis

Criterion: Safety from side effects when ConcernExtractor suggests statement reaggregation

One of the main objectives of ConcernExtractor is to be able to determine if a crosscutting concern can be extracted into a standalone method. The JDT *Method Extract* refactoring can only be applied to a series of consecutive code statements. However, we need to also consider method clusters that belong to a refactorable crosscutting concern, but are interleaved with other statements. In order to extract these concerns with interleaved statements (hereby referred to as *non-consecutive clusters*), the ConcernExtractor uses flow analysis

to verify if the executed order of the code statement can be reordered (*reaggregated*), and reorder the statements that belong to the crosscutting concern so that they can be extracted.

Since it is impractical to perform full static analysis for verifying if the state of each object would change during reaggregation, we only apply flow analysis in the scope of the cluster. We verify that the objects being called, passed, or returned in the statement that we want to reorder, are not being called or passed in the subsequent statements. However, we do not recursively inspect the internals of the methods to check for state changes in passed objects. This use of flow analysis exposes a potential deficiency, that we could not automatically verify if the reaggregation can produce undesired side-effects.

To the evaluate the effectiveness and the accuracy of statement reaggregation, we apply flow analysis to each cluster group and categorize the clusters in each cluster group into the following categories:

Consecutive clusters: Clusters where seed methods are executed consecutively.

Non-consecutive clusters: Clusters where the seed methods statements are not contiguous.

Reaggregable clusters: Non-consecutive clusters; however, the flow analysis suggests that the seed methods statements can be safely *reaggregated* without causing side-effects.

Non-reaggregable clusters: Non-consecutive clusters with statements that cannot be reaggregated safely.

False positive: Non-consecutive clusters that are determined by our flow analysis as *reaggregable*; however, reordering the statements would introduce unintended behaviour or errors, therefore is not desirable to reaggregate the statements.

Tables 4.3 and 4.4 describe the data of our analysis for clusters groups that contain more than two and only two seed method statements, respectively. We shows the number of consecutive clusters alongside the non-consecutive clusters to emphasize the frequency of the non-consecutive clusters in a cluster group.

From the results of the evaluation, we made the following observations.

Table 4.3: List of clusters that contain more than or equal to 3 seed methods

Target System	Concern Methods [a]	Consecutive Clusters	Non-Consecutive Clusters	Flow Analysis		
				Reaggregable [b]	Non-Reaggregable	False Positive

JBossAOP	`make()`
	`setmodifier()`
	`addMethod()`
		9	2	2	0	2

JBossAOP	`make()`
	`setmodifier()`
	`addMethod()`
		17	1	1	0	0

jEdit	`setLocationRelativeTo()`
	`pack()`
	`setVisible()`
		8	1	1	0	1

jEdit	`setLocationRelativeTo()`
	`pack()`
	`setVisible()`
		2	1	1	0	1

Freemind	`startTransaction()`
	`executionAction()`
	`endTransaction()`
		29	3	3	0	0

Freemind	`startElement()`
	`childAsURIs()`
	`endNamespaceDecls()`
	`childAsAttributes()`
	`endAttributes()`
	`childAsElementBody()`
	`endElement()`
		4	0	0	0	0

ArtOfIllusion	`getObject()`
	`setVertexPosition()`
	`objectChanges()()`
		1	5	2	0	0

ArtOfIllusion	`setVertexPosition()`
	`updateImage()`
	`objectChanges()()`
		17	0	0	0	0

Ant	`CommandLine.ctor()`
	`setExecutable()`
	`setValue()()`
		26	0	0	0	0

Ant	`Execute.ctor()`
	`setAntRun()`
	`setCommandLine()()`
	`setWorkingDirectory()()`
		3	2	2	0	0

Ant	`Execute.ctor()`
	`setAntRun()`
	`setCommandLine()()`
	`setWorkingDirectory()()`
		3	0	0	0	0

[a] Since we only include the method name in the table, there are concerns that seem to be identical. In fact, the signatures of these methods are all different.

[b] Total number of clusters flagged *reaggregable* by the ConcernExtractor

Table 4.4: List of clusters that contain only 2 seed methods

Target System	Concern Methods [a]	Consecutive Clusters	Non-Consecutive Clusters	Flow Analysis		
				Reaggregable [b]	Non-Reaggregable	False Positive
JBossAOP	`make()` `addMethod()`	16	1	1	0	1
JBossAOP	`make()` `addMethod()`	8	0	0	0	0
JBossAOP	`setModifiers()` `addField()`	21	0	0	0	0
JBossAOP	`setModifiers()` `addField()`	6	0	0	0	0
jEdit	`openNodeScope()` `jjtreeOpenNodeScope()`	38	0	0	0	0
jEdit	`openNodeScope()` `jjtreeOpenNodeScope()`	13	0	0	0	0
jEdit	`closeNodeScope()` `jjtreeCloseNodeScope()`	51	0	0	0	0
FreeMind	`text()` `childAsAttributes()`	148	0	0	0	0
ArtOfIllusion	`updateImage()` `objectChanged()`	17	0	0	0	0
ArtOfIllusion	`updateImage()` `objectChanged()`	11	0	0	0	0
Ant	`setValue()` `setCommandLine()`	1	8	8	0	0
Ant	`setValue()` `setWorkingDirectory()`	10	0	0	0	0
Ant	`setCommandLine()` `setWorkingDirectory()`	10	0	0	0	0
Ant	`setCommandLine()` `log()`	7	0	0	0	0
Ant	`createTempFile()` `deleteOnExit()`	4	0	0	0	0
Ant	`createTempFile()` `deleteOnExit()`	4	0	0	0	0
Ant	`createTempFile()` `deleteOnExit()`	2	0	0	0	0

[a] Since we only include the method name in the table, there are concerns that seem to be identical. In fact, the signatures of these methods are all different.

[b] Total number of clusters flagged *reaggregable* by the ConcernExtractor

- The number of non-consecutive clusters is generally much less than the consecutive clusters within a cluster group, with the exception of cluster groups that contain only one consecutive cluster.

- Most non-consecutive clusters are considered reaggregable by flow analysis. However, flow analysis should not be considered as 'safe' due to its low but significant inaccuracy.

- For a group of method clusters that are considered isomorphic at least one of the clusters must be consecutive, otherwise it is not feasible to determine if the crosscutting code is a consistent behavior concern, and if it makes sense to extract it to aspect. However, we observed from the results, there are many cluster groups that contain only one consecutive cluster. The prevalence of this type of cluster group suggests that:

    1. The cluster that contains consecutive crosscutting code may be a code defect or,
    2. The group is incorrectly labelled as crosscutting (false positive) when the methods in the cluster do not belong to the same crosscutting concern.

- We noticed that in all cases where reaggregation introduces unexpected side-effects, the concern group only contains 1 or 2 non-consecutive clusters compared with an overwhelming ratio of consecutive clusters in that cluster group. This property indicates that the particular non-consecutive cluster is a special case of that concern, when the crosscutting statements need to be interleaved with other method calls in the execution.

From the observations stated above, we conclude that there are several factors which diminish the percentage accuracy of flow analysis in our evaluation. However, given the scarcity of non-consecutive clusters within a concern group, the scalability issue of the analysis may not be as critical as we initially assumed and a bytecode-based side effect

Table 4.5: Summary of flow analysis results

Cluster Type	% $\frac{Reaggregable\ clusters}{Non-consecutive\ clusters}$	% $\frac{Mis-flagged\ clusters\ by\ FA}{Reaggregable\ clusters}$
Clusters with $\geq$3 statements	80 %	33 %
Clusters with 2 statements	100 %	16 %

analysis using tools such as the Soot Framework[2] can be substituted for better accuracy. Moreover, aspect refactoring tools generally requires a user to manually determine if a candidate crosscutting concern should be refactored to aspect. Thus, user intervention is unavoidable in most cases. The ConcernExtractor reduces the overhead in this process by mapping out crosscutting code and refining the scope of intervention by the user.

4.5 Question 3: Concern extraction

Criterion: the necessity of concern extraction to isolate the identified crosscutting clusters.

The ConcernExtractor contains an annotated source code viewer that enables a user to quickly browse to the location of the cluster in the source file. Hence, the user can easily navigate to the method and the control block that contains the target cluster, and check if the cluster is located at the **start/end** of the method body, or if it is nested by a `try` block, a `for` loop, or other control structure

With the help of the ConcernExtractor, we can estimate the difficulty in directly refactoring a segment of code into aspect, based on the control flow and the structures of its surrounding code. The basic principle of refactoring is "altering internal structure (of existing code) without changing its external behavior" [13]. Already, we have seen that aspect refactoring sometimes requires loosening this behavioral preservation criteria in order to refactor the crosscutting concern into aspect (see section 3.4, [16]). However, we

[2]See http://www.sable.mcgill.ca/soot.

(or the developer) should expect that the control flow of the execution before and after the refactoring to remain largely the same — the aspect code should be woven at the same location before it was refactored.

On the other hand, a refactoring that preserves the same control flow is often not possible, due to the limitations in the pointcut languages. For instance, AspectJ only provides pointcut designators that intercept join points at certain places in the control flow, such as the start of a method call, or before an instance variable is accessed (to see the complete list of pointcut descriptors, please refer to the AspectJ Programming Guide[3]).

Our concern extraction approach circumvents this problem by creating new methods for isolating the crosscutting code. The extracted methods that contain the isomorphic clusters will have a common method signature. To extract the crosscutting clusters into a single aspect, a user can create a pointcut expression that intercepts the join points of these methods, and then consolidate the concern to aspect by extracting the content of the methods into **one** single advice. In this case, the extracted methods are empty and merely placeholders for the pointcut. During compilation (or run-time), the aspect compiler (weaver) can weave the crosscutting code in the advice back at its invocation location, and replicate the pre-extraction behavior.

However, does this approach offer a substantial increase in refactoring opportunities? Moreover, out of the refactorable crosscutting clusters discovered by the ConcernExtractor, how many clusters do not need the extraction to a standalone method and can be directly refactored into aspect? In this section, we attempt to answer this question by assessing the numbers of clusters in a refactorable crosscutting concern that are *directly refactorable*: clusters that can be immediately refactored into an aspect using a simple pointcut expression. To help us to obtain this value, we devise an equation based on the location of a crosscutting method cluster relative to its containing body.

[3]See http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html.

$$\text{No. of refactorable clusters} = \left\{ \begin{array}{c} \text{No. of clusters} \\ \text{at the start/end} \\ \text{of method body} \end{array} \right\} - \left\{ \begin{array}{c} \text{No. of clusters} \\ \text{inside } \textbf{try} \text{ block} \end{array} \bigcup \begin{array}{c} \text{No. of clusters} \\ \text{inside a loop block} \end{array} \bigcup \begin{array}{c} \text{No. of clusters inside} \\ \textbf{conditional} \text{ block} \end{array} \right\}$$

Constraint 1: Only method clusters that are located at the beginning or the end of a method can be considered as a candidate of a concern that is directly refactorable.

In general, code snippets that are located at the beginning or the end of a method declaration can be extracted into aspects using a simple pointcut expression that consists a `call` or `execution` pointcut designator. However, if the cluster is located in the middle of the containing method body, creating a pointcut expression that intercepts the join points before or after the cluster involves at least two pointuct designators: a control flow-based designator (i.e. `cflow` or `cflowbelow`) that locates the containing method body, and a method call designator (i.e. `call` or `execution`) using the statement above or below the cluster. This type of pointcuts must be carefully crafted to avoid unintended side effects, and a side-effects-proof pointcut must account for all the variations, which adds additional complexity to the pointcut expression. Therefore, we do not consider the cluster located in the middle of its containing method body to be directly refactorable.

Constraint 2: Clusters in a `try` *block should not be refactored.*

To correctly refactor a segment of code inside a `try` block to an *advice* in the aspect class requires that:

1. The advice catches any exceptions the statements in the target code segment may throw, but this approaches implies that the `catch` block must also be refactored into the aspect; or

2. The advice re-throws any exception thrown by the code segment. However, since the aspect code will be called before of after the actual method code(depending on the type of advice used), any exception thrown by the aspect code will not be caught

by any `try` blocks within the method body. The enclosing method will need to declare in its signature any exception the aspect may throw; otherwise it will cause a compile-time/run-time error. Adding new exceptions to a new method changes the control flow of the system and therefore is not an acceptable refactoring.

Constraint 3: Clusters inside a loop are not considered refactorable.

AspectJ does not have support for a pointcut designator that picks out join points at a `for/while` loop clause.

Constraint 4: Clusters inside a conditional statement (e.g. `if/else/switch`) *are also not considered refactorable.*

Although AspectJ provides an `if` *conditional* pointcut designator, it can only be used as a conditional for another join point (i.e. to evaluate if a join point should be weaved based on another conditional expression), but it cannot be used to intercept the join point at an `if` conditional statement.)

In the previous sections, we identified groups of *refactorable crosscutting clusters* using the ConcernExtractor. In this section, for each group of refactorable crosscutting clusters, we manually inspect the position of each cluster relative to its containing block and method body, to determine whether it is:

a. Located the the beginning or the end of the method body, and

b. Contained within a loop, or an `if` statement.

Using the formula we described above, we can determine the number of *directly refactorable* clusters from the results. In our evaluation, we limit the target to only groups of clusters that contain at least **three** statements.

Table 4.6 depicts a summary of the positions of method clusters relative to each of their declaring method. The concerns in the table are essentially identical to the concerns

Table 4.6: Summary of positions of method clusters relative to their declaring method bodies

System	Concern	Clusters at **begin/end** of method body	Clusters inside **try** block	Clusters inside **for** block	Clusters inside **if** block	Directly refactorable clusters	Extractable clusters
JBossAOP	Add Method I	17	12	0	1	4	26
JBossAOP	Add Method II	9	0	0	0	9	9
ArtOfIllusion	Reset Vertex I	3	0	0	0	3	9
ArtOfIllusion	Reset Vertex II	3	0	0	0	3	6
Ant	Run Ant Command I	5	5	0	0	0	5
Ant	Run Ant Command II	3	3	0	0	0	3
FreeMind	XML Serialize	4	0	0	4	0	4
FreeMind	Transaction	32	22	0	1	9	32

that were listed in Table 4.3. However, for some of the concerns, the name of their seed methods overlap. These methods only differ in the number and the types of the parameters. In order to distinguish between different concerns that have the same functionalities, we abbreviate the labels of the concern and denote the concerns with a number (i.e. I & II) to show the difference.

From the result, we can observe that the number of *extractable* clusters are substantially more than the number of clusters that are directly refactorable. However, the result also provides additional insights about the adequacy of AspectJ as the target aspect language for refactoring.

One of the original expectations from the results is that the number of clusters located at the start or the end of a method is scant, compared to the number that are in the middle of the method body. Nevertheless, among the refactoring crosscutting concerns that serve as the subject of this evaluation, most of them contains a significant proportion ($> 50\%$) of method clusters that are located at the start or end of method bodies. Although the results do not imply that AspectJ provides adequate pointcut designators for refactoring, it suggests that the beginning or the end of a method is a common location for crosscutting concerns, and these major refactoring opportunities can be captured using simple pointcut provided by AspectJ.

Also, for a system with significant size, code for exception handling should be prevalent, and it will be common to find method code that is totally covered by a `try` block. We observed some instances of a large `try` block in the clusters from JBossAOP and FreeMind, and the evaluation shows that it is the most significant block pattern that prevents the method clusters from being directly refactorable. One common exception handler scenario that we found in the target systems involves a candidate cluster located at the beginning of a huge `try` block that encapsulates the rest of the method code. The *exception handling* pattern [22] is an AspectJ refactoring that is capable of completely extracting the exception handling component from a method body into an aspect. One area for future work may be the evaluation of the applicability of such patterns for recovering more opportunities for refactoring the crosscutting clusters inside a `try` block.

Chapter 5
Conclusions

We proposed a technique for bridging the gap between aspect-mining and AO refactoring, and implemented our approach in a tool called ConcernExtractor. ConcernExtractor is capable of assessing the results of an aspect-mining tool to identify the refactorable crosscutting concerns in the system. It also supports applying a series of code transformations to extract these crosscutting code segments into new standalone methods using a common method signature. As a result, the inherent regularity of the crosscutting is captured explicitly in the structure of the code, which can be leveraged to extend the code with additional aspects.

We evaluated our approach by applying it to five medium-size open-source systems. From our evaluation, we found that the number of crosscutting instances that are considered refactorable by ConcernExtractor are scarce, compared to the size of the target systems. We conclude that AOP refactoring has limited potential for improving the modularity in an existing system that lacks initial considerations for crosscutting concerns.

In conclusion, we believe the major contribution of this research is to provide mechanism to filter the results from aspect-mining, and to identify the program elements that truly constitute a crosscutting concern that can be refactored into aspect. For existing systems, we believe that it is essential for a programmer to be aware of the existence of such crosscutting concerns, regardless of the necessity of refactoring to aspect.

We developed the ConcernExtractor because current aspect mining techniques do not always produce results that we can intuitively recognize as a crosscutting concern, and these candidates do not equate to code fragments that are refactorable to aspects. To ensure a minimal set of requirement for refactorable concerns, we reduce the refactoring candidates to a small set of constructs in the Java language (i.e. method invocation statements) that can be mapped to AspectJ. An interesting area for future research would be expanding the candidates from method invocation statements to more complex expressions that contain the seed methods, and explore the required AspectJ constructs to refactor them.

We also propose that the concept of method cluster be further developed. Currently, a method cluster is limited to a contiguous statements of method invocation. However, there are other scenarios of crosscutting, such as methods that are called at the beginning and end of a method, or inside a `try-finally` block, that are not captured because of the requirement of contiguousness. Method cluster could be expanded to incorporate these crosscutting concerns.

The current ConcernExtractor is intended to facilitate AOP refactoring using aspect-mining results. To further validate the concern extraction mechanism and the usefulness of aspect refactoring, we suggest that the ConcernExtractor should incorporate aspect-refactoring functionality and become a comprehensive refactoring framework. It will help us to conduct comprehensive aspect refactoring studies using aspect-mining results.

Bibliography

[1] P. Anbalagan and T. Xie. Automated inference of pointcuts in aspect-oriented refactoring. In *Proceedings of the 29th International Conference on Software Engineering*, pages 127–136, 2007.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 199–211, 2007.

[7] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 37–46, 2005.

[8] M. Bruntink, A. van Deursen, T. Tourwé, 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.

[9] 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.

[10] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[11] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 254–263, 2007.

[12] A. v. Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03)*. University of Waterloo, Canada, 2003.

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

[14] B. Ganter, G. Stumme, and R. Wille, editors. *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*. Springer, 2005.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[16] J. Hannemann. Aspect-oriented refactoring: Classification and challenges. In *Proceedings of 5th Workshop on Linking Aspect Technology and Evolution (LATE)*, 2005.

[17] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In *Proceedings of Workshop on Advanced Separation of Concerns, International Conference on Software Engineering (ICSE 2001)*, 2001.

[18] 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.

[19] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 135–146, 2005.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of ACM*, 44(10):59–65, 2001.

[21] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. 1997.

[22] R. Laddad. *Aspect Oriented Refactoring*. Addison-Wesley Professional, 2008.

[23] C. Lindig. *Fast Concept Analysis*. Shaker Verlag, August 2000.

[24] M. Marin. Refactoring JHotDraw's Undo concern to AspectJ. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE).*, 2004.

[25] 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.

[26] 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.

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

[28] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer, 2008.

[29] 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.

[30] 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.

[31] A. O'Connor, M. Shonle, and W. Griswold. Star diagram with automated refactorings for Eclipse. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 16–20, 2005.

[32] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[33] M. P. Robillard and F. Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 65–69, 2005.

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

[35] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 3–14, 2006.

[36] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.

[37] 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.

[38] I. Yuen and M. Robillard. Bridging the gap between aspect mining and refacctoring. In *Proceedings of 6th Workshop on Linking Aspect Technology and Evolution (LATE)*, 2007.

[39] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 130–139, 2003.

[40] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 226–238, 2007.