

2018

Metrics for Aspect Mining Visualization

Gisle J. Jorgensen

Nova Southeastern University, jjorgensen2311@gmail.com

This document is a product of extensive research conducted at the Nova Southeastern University [College of Engineering and Computing](#). For more information on research and degree programs at the NSU College of Engineering and Computing, please click [here](#).

Follow this and additional works at: https://nsuworks.nova.edu/gscis_etd

 Part of the [Computer Sciences Commons](#)

Share Feedback About This Item

NSUWorks Citation

Gisle J. Jorgensen. 2018. *Metrics for Aspect Mining Visualization*. Doctoral dissertation. Nova Southeastern University. Retrieved from NSUWorks, College of Engineering and Computing. (1048)
https://nsuworks.nova.edu/gscis_etd/1048.

This Dissertation is brought to you by the College of Engineering and Computing at NSUWorks. It has been accepted for inclusion in CEC Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact nsuworks@nova.edu.

Metrics for Aspect Mining Visualization

by


Gisle J. Jorgensen

A dissertation report submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Information Systems

College of Engineering and Computing
Nova Southeastern University

2018

We hereby certify that this dissertation, submitted by Gisle Jorgensen, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.



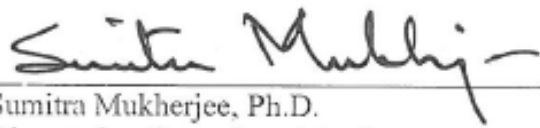
Francisco J. Mitropoulos, Ph.D.
Chairperson of Dissertation Committee

July 14, 2018
Date



Junping Sun, Ph.D.
Dissertation Committee Member

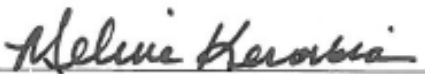
July 16, 2018
Date



Sumitra Mukherjee, Ph.D.
Dissertation Committee Member

July 16, 2018
Date

Approved:



Meline Kevorkian, Ed.D.
Interim Dean, College of Engineering and Computing

July 16, 2018
Date

College of Engineering and Computing
Nova Southeastern University

2018

An Abstract of a Dissertation Submitted to Nova Southeastern University

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Metrics for Aspect Mining Visualization

by
Gisle J. Jorgensen

2018

Aspect oriented programming has over the last decade become the subject of intense research within the domain of software engineering. Aspect mining, which is concerned with identification of cross cutting concerns in legacy software, is an important part of this domain. Aspect refactoring takes the identified cross cutting concerns and converts these into new software constructs called aspects. Software that have been transformed using this process becomes more modularized and easier to comprehend and maintain. The first attempts at mining for aspects were dominated by manual searching and parsing through source code using simple tools. More sophisticated techniques have since emerged including evaluation of execution traces, code clone detection, program slicing, dynamic analysis, and use of various clustering techniques. The focus of most studies has been to maximize aspect mining performance measured by various metrics including those of aspect mining precision and recall. Other metrics have been developed and used to compare the various aspect mining techniques with each other. Aspect mining automation and presentation of aspect mining results has received less attention. Automation of aspect mining and presentation of results conducive to aspect refactoring is important if this research is going to be helpful to software developers. This research showed that aspect mining can be automated. A tool was developed which performed automated aspect mining and visualization of identified cross cutting concerns. This research took a different approach to aspect mining than most aspect mining research by recognizing that many different categories of cross cutting concerns exist and by taking this into account in the mining process. Many different aspect mining techniques have been developed over time, some of which are complementary. This study was different than most aspect mining research in that multiple complementary aspect mining algorithms was used in the aspect mining and visualization process.

Acknowledgements

I would like to thank Dr. Mitropoulos (chair) for all his helpful feedback at the various stages during the dissertation process. I would also like to thank the committee members Dr. Mukherjee and Dr. Sun for all their helpful feedback and the document revision suggestions.

Table of Contents

Abstract ii

Acknowledgements iii

List of Tables vii

List of Figures viii

Chapters

1. Introduction 1

Background 1

Problem Statement 4

Dissertation Goal 4

Research Questions 5

Relevance and Significance 5

Barriers and Issues 7

Limitations and Delimitations 8

Definition of Terms 9

Summary 10

2. Review of the Literature 11

Introduction 11

Aspect Mining Techniques 11

Visualization of Aspect Mining Results 16

3. Methodology 18

Overview of Research Methodology 18

Aspect Mining and Visualization Tool 18

Aspect Mining Using Model Based Clustering 22

Aspect Mining Clone Detection	24
Aspect Mining Using Event Tracing	26
Synthesis of Aspect Mining Results	29
Visualization of Cross Cutting Concerns in Source Code	31
Resource Requirements	32

4. Results 34

Data Generation	34
Data Analysis	45
Summary of Results	56

5. Conclusions, Implications, Recommendations, and Summary 57

Conclusions	57
Implications	57
Recommendations	58
Summary	59

Appendices 62

A. Aspect Mining and Visualization Tool Architecture	62
B. Aspect Mining and Visualization Tool Application Components	63
C. Aspect Oriented Programming Artifacts for Method Execution Trace Log	65
D. Fan-In Results File Excerpt	70
E. Aspect Mining Results DTD	71

References 72

List of Tables

Tables

1. Ordered Method Call Analysis Results 46
2. Code Clone Analysis Results 47
3. Methods in JHotDraw with Highest Fan-In Values 47
4. Cross Cutting Concern as Interface Analysis Results 48
5. Identified Calls in Clones Cross Cutting Concern 50
6. Identified Calls at the Beginning of Methods Cross Cutting Concern 51
7. List of Methods Calling CollectionsFactory.createList Method at the Beginning 52

List of Figures

Figures

1. Aspect Mining and Visualization Tool User Interface 20
2. Aspect Mining and Visualization Tool 21
3. Event Trace Information from Fictitious Program Executions 27
4. Venn Diagram Representing Identified Sets of Cross Cutting Concerns 29
5. Fully Duplicated Code for Methods in Different Classes 36
6. Duplicated Code at the Beginning of Methods, Candidate for Before Advice 37
7. Duplicated Code at the End of Methods, Candidate for After Advice. 38
8. Method Candidate for Around Advice 39
9. Identification of Ordered Method Call Cross Cutting Concerns with Inside Relation 42
10. Identification of Ordered Method Call Cross Cutting Concerns with Inside Relation 42
11. Sample XML Representation of Aspect Mining Results 45
12. Identification of Cross Cutting Concern as Interface 49
13. Example of Frequent Calls in Clones Cross Cutting Concern 50
- 14: Example of Calls at the End of Methods Cross Cutting Concern 53
15. User Selects Cross Cutting Concerns Categories to Mine for 53

- 16. User Clicks Cross Cutting Concern Details Link to get more Information 54
- 17. User Clicks Cross Cutting Concern Details Link (clicked line in red) 55
- 18. The Remaining Detail for the Calling Methods is Displayed when Scrolling Down 55
- 19. Aspect Mining and Visualization Application (AMV) Architecture 62
- 20. Aspect Mining and Visualization Tool Application Components 63
- 21. Configuration of MethodExecutionTrace Aspect 65
- 22. Sample Ignore and Include Calling Relations from method_execution_trace.log 66
- 23: Inside Relations Identified by Automated Analysis of Generated Execution Trace Log 67
- 24. Shade Plugin Used to Create Executable jar File for Method Tracing 68
- 25. Log4j Configuration Capturing Trace Log in C:/log/method_execution_trace.log 69
- 26. Fan-In Results File Excerpt 70
- 27. DTD for XML Representing the Aspect Mining Results 71

Chapter 1

Introduction

Background

Functional business requirements representing core concerns can be successfully implemented using object oriented programming techniques. Nonfunctional requirements such as logging, transaction management, persistence, and error handling do not fit well into the object oriented model (Bruntink, Deursen, Engelen, & Tourwe, 2005). Code representing these concerns are often tangled in with code implementing core concerns when traditional software engineering principles are applied (Cojocar, & Czibula, 2008). Further, the code implementing cross cutting concerns are scattered throughout the code base in modules implementing core concerns. Legacy systems where the code base is composed of modules that implements both core and cross cutting concerns tend to be complex, hard to maintain, and difficult to enhance (Cojocar, Czibula, & Czibula, 2009). Aspect Oriented Programming (AOP) a relatively new software engineering invention which makes it possible to separate the implementation of cross cutting concerns from that which implements the core concern (Martin, Deursen, & Moonen, 2004). The cross cutting concerns are implemented in software constructs called aspects when AOP is used. The aspects are, depending on the specific implementation of the AOP system, woven into the code base of the executable unit at compile time, load time, or execution time.

The design of legacy systems can be transformed into one in which core concerns and cross cutting concerns are implemented in separate modules. The first step in this process is to identify the aspect seeds in the legacy code (Shepherd, Gibson, & Pollock, 2004). The aspect seeds are discovered in the legacy code through the process of aspect mining. Much research has

been devoted to aspect mining and many aspect mining techniques have been developed. The quality of the aspect mining techniques has improved over the years which has resulted in better results for aspect mining recall (the percentage of cross cutting concerns discovered compared to all cross cutting concerns in the software) and aspect mining precision (the percentage of actual cross cutting concerns retrieved compared to all instances retrieved) (Moldovan, & Serban, 2006b). Recent clustering based aspect mining techniques are more automated than earlier techniques. Metrics have been developed to measure and compare various aspect mining clustering techniques (Rand McFadden, & Mitropoulos, 2013b).

This research will focus on visualization metadata which will be used to depict cross cutting concerns in source code after this has been mined for cross cutting concerns. The goal is to make aspect mining results practically available to the software developer through visualization so that legacy software can be effectively transformed to aspect oriented software. An aspect mining visualization prototype tool will be developed to show the usefulness of the aspect mining visualization metadata by demonstrating how these are used to depict cross cutting concerns for aspect refactoring purposes in source code.

Cross cutting concerns comes in many different forms. Still most aspect mining research treats them as a whole. This is also one of the reasons why less effective aspect mining and aspect refactoring tools have been developed to date. The focus of this research is to identify what category each of the located cross cutting concerns belong to and then graphically represent these. Each visualized cross cutting concern will be annotated indicating which cross cutting concern category they belong to. This will help the software developer in the design decisions he must make when transforming cross cutting concerns by, (1) where and how to implement point cuts, (2) deciding on type of advice, and (3) the implementation of aspects.

The different categories of aspects that the tool will be used to mine for include: (1) Ordered Method Call: These calls that are always called relatively to other method calls (Shepherd, Palm, & Pollock, 2005). (2) Code Clone: These are cross cutting concerns characterized by identical or very similar code that is in many different parts of the application (Bruntink, Deursen, Engelen, & Tourwe, 2005). Transaction demarcation code falls into this category. This code demarcates transaction begin, commit, and rollback. This code can be encapsulated in a transaction aspect which can then be advised using point-cut expressions. (3) Unique Class Fan In: This concern is characterized by many unique classes that makes calls to one specific method (Martin, Deursen, & Moonen, 2004). (4) Calls In Clones: This concern is characterized by code clones scattered in the application. The cloned code contains frequently calls to a method M which is a good candidate for a cross cutting concern that can be aspectized (Bruntink, Deursen, Engelen, & Tourwe, 2005). (5) Calls at Beginning and End of a Method: Calls placed at the beginning and at the end of a methods are often implementing a cross cutting concern. One such example is logging for debugging purposes when the façade pattern is used. Developers often place log statements at the beginning and at the end of façade APIs to trace execution flow entry and exit points. Façade is a commonly used design pattern in client server application architectures. AOP can also be implemented for design patterns (Hannemann, & Kiczales, 2002; Laddad, 2003). The following categories of cross cutting concerns that relates to design patterns will be investigated: (6) Persistence, (7) Command, (8) Observer, and (9) Decorator. Aspect visualization metadata describing these will be designed and used when mining for cross cutting concerns. The last category of cross cutting concerns that will be investigated is: (10) Cross-cutting concerns that are represented as an interface with method implementations that appear to be consistent (Shepherd, & Pollock, 2005).

Problem Statement

The problem addressed by this research is that the various categories of cross cutting concerns cannot be automatically mined and visualized in legacy code. There are three reasons why this problem has not been solved: (1) Current aspect mining techniques are focused on identification of cross cutting concerns in general and not on categories of cross cutting concerns. (2) Most aspect mining techniques are focused on improving aspect mining performance. This study will visualize the identified cross cutting concerns. (3) Most aspect visualization tools are focused on showing aspects in software where cross cutting concerns already have been implemented as aspects.

Dissertation Goal

The goal of this research is to automatically find aspects using aspect mining techniques based on clustering, cloning, and execution traces. Each cross cutting concern will automatically be identified and visualized using the prototype that will be built as part of this study. Each visualized cross cutting concern will be annotated with the category it belongs to. The visualized cross cutting concerns represents aspect mining refactoring candidates. Aspect mining benchmark software will be used when mining for aspects.

Research Questions

- (1) Is it possible to capture enough information using aspect mining clustering, cloning, and analysis of execution traces to identify and categorize cross cutting concerns in legacy source code?
- (2) Can visualization metadata be created to depict cross cutting concern in source code and identify which category of cross cutting concerns they belong to?
- (3) Can visualization metadata in XML format be used to describe cross cutting concerns so that software developers can refactor these into aspects?
- (4) How successful is this research at identifying the cross cutting concerns in the benchmark software?

Relevance and Significance

A large body of software systems falls under the category of legacy systems. These are software systems that have lived beyond the first generation of programmers (Kontogiannis et al. 2003). The cost of maintaining and evolving these systems are disproportionally larger than the amount of code changed or the resulting change in software behavior (Griswold, Yuan, & Kato, 2001). One cause of difficulty in maintaining these systems is lack of modularity. Business requirements representing core concerns of the system are usually successfully implemented according to well established object-oriented design patterns and principles. System requirements such as logging, transaction management, and persistence are cross cutting and require implementations across core concerns if only object-oriented software principles are applied during software construction.

Aspect oriented programming techniques facilitate implementation of cross cutting concerns in modules called aspects. These can be invoked using point-cuts and advice (Czibula, Cojocar, & Czibula, 2009b). Software systems that are easier to maintain and evolve can be constructed by encapsulating core concerns in modules based on object oriented principles and by encapsulating cross cutting concerns in modules using aspect oriented techniques. Legacy systems that contain cross cutting concerns can be reverse engineered to aspect oriented systems. This process has two steps (Zhang, Guo, & Chen, 2008). The first step is to locate the cross cutting concerns or the aspects seeds. This is the process of aspect mining. Much research has been focused on aspect mining. The second part of this problem is to transform the identified aspect candidates into actual aspects (Binkley, Ceccato, Harman, Ricca, & Tonella, 2005; Monteiro, & Fernandes, 2004). This is the process of aspect refactoring.

Less research has been focused on aspect refactoring. Most of the research in this area is still immature and the proposed techniques have not lived up to their expectations (Mens, Kellens, & Krinke, 2008). Aspect mining has come far in automating the process of identifying aspect candidates. (Rand McFadden, & Mitropoulos, 2012), but little progress has been made using these aspect mining results in the next step of aspect refactoring. Tools that automates large parts of the refactoring process is needed if reverse engineering large software systems for aspect oriented programming is ever going to become practically viable. Most tools that have been developed are either lexical (Hanneman, & Kiczales, 2001; Griswold, Yuan, & Kato, 2001) or exploratory (Janzen, & Volder, 2003; Robillard, & Murphy, 2002). Neither of these approaches are automatic. The tools depend on user familiarity with the software and that the user can find aspects seeds in the legacy software that is being mined.

Shepherd, Gibson, and Pollock, (2004) developed Timna which is a tool for aspect mining and analysis. This tool is based on Fan-in analysis for the aspect mining part and manual tagging for identification of cross cutting concerns in the legacy code. While this tool has more support for automation than preexisting tools it still relies on Fan-in analysis for identification of aspect mining candidates. More sophisticated clustering techniques have since been used for better cross cutting concern identification and automation. Another shortcoming of this tool and prior attempts at automating aspect mining and visualization is that user involvement is required in the identification phase of aspect mining candidates. The tool that will be developed as part of this study will identify aspect candidates automatically by using a combination of aspect mining techniques based on model based clustering (Rand McFadden, & Mitropoulos, 2012), program dependency graph based clone detection (Komondoor, & Horwitz, 2001; Krinke, 2001), and event based program tracing (Breu, & Krinke, 2004). The usability of the aspect mining visualization metadata will be demonstrated with the aspect mining visualization prototype.

Barriers and Issues

JHotDraw (v.5.4b1) is the most commonly used benchmark software for aspect mining algorithms. This medium sized application has well defined cross cutting concerns, but may not be representative of larger software systems (Rand McFadden, & Mitropoulos, 2013). Another shortcoming with JHotDraw is that the software does not contain many of the different categories of cross cutting concerns that will be mined for. JHotDraw will be used for this study because many other aspect mining and refactoring studies have used this software and the results can therefore be compared to that of other studies. JHotDraw is a very clean implementation since it was built to showcase specific design patterns and principles. JHotDraw is implemented

in more than 12,000 lines of code. JHotDraw will be seeded with code snippets as necessary when mining for certain categories of cross cutting concerns. Cross cutting concerns categories that builds upon code clones will be seeded in JHotDraw since this software does not because of its clean implementation contain much clones. This will allow for a complete set to cross cutting concerns representing all categories relevant to this study will be present in the software to be mined.

Limitations and Delimitations

JHotDraw will be used as input for the Aspect Mining and Visualization Tool. JHotDraw is the most commonly used benchmark software for aspect mining studies and is therefore good for measuring relative effectiveness of aspect mining algorithms. The software has a known set of cross cutting concerns which makes it possible to obtain values for metrics such as precision and recall. The software is highly structured and does therefor not contain much code duplication. Some seeding will therefore be performed so that it will be possible to find results when mining for cross cutting concerns that are based on code clones. JHotDraw is a medium sized software system and will not give a good measure for how well the aspect mining techniques will perform for large or very large software systems.

Definition of Terms

Aspect	Functionality implemented as separate modules that can be executed as part of core modules during program execution.
Aspect Mining	Identification of cross cutting concerns or aspect candidates in the source code of an existing software system.
Aspect Refactoring	The transformation of the identified cross cutting concerns into aspects.
Abstract Syntax Tree	A tree representation of the abstract syntactic of source code written in a programming language.
Aspect Mining Recall	The fraction of cross cutting concerns discovered compared to all cross cutting concerns in the software.
Aspect Mining Precision	The fraction of actual cross cutting concerns retrieved compared to all instances of cross cutting concerns in the software.
Code Scattering	When a single functionality is implemented in multiple modules.
Code Tangling	When code to handle multiple concerns is interleaved in the same module.
Core Concerns	The main business functionalities of the system.
Code Clone	Sequences of duplicate code, or sequences of code which with the same input produces the same output.
Cross Cutting Concern	Functionalities that cuts across multiple modules.
Design Patterns	A general solution to a design problem that occurs repeatedly in many software systems.
Dynamic Analysis	Evaluation of program by executing data in real time.
Execution Traces	Trace information, such as method entry and method exit, recorded when control flows through a program during program execution.
Point Cut	A program construct that selects join points and selects context at those points.
Program Dependency Graph	Graph representing program control and data dependencies.

Static Analysis	Examination of code without executing the program.
Separation of Concerns	Design principle for separation of source code into modules so that each represents a distinct concern.

Summary

Object oriented programming has played a significant role in reducing the complexity of software systems. Core concerns can be implemented in separate cohesive core modules where each module has a single responsibility. Cross cutting concerns such as logging, transaction and error handling are required across core modules and are therefore better implemented as aspects. Mining for cross cutting concerns is the first step when refactoring legacy software into an aspect oriented system a software system. This study uses two static and one dynamic aspect mining technique when performing automated aspect mining. The cross cutting concerns are assigned to ten different aspect mining categories and visualized using a prototype tool that will be developed for this study.

Chapter 2

Review of the Literature

Introduction

Literature from aspect mining and aspect visualization is reviewed. Aspect mining literature is reviewed since clustering and other aspect mining techniques will be used to identify cross cutting concern in source code. Aspect visualization literature is reviewed since the identified cross cutting concerns will be visualized.

Aspect Mining Techniques

Early aspect mining techniques were manual and tool based. One of the first attempts to localize concerns that could be encapsulated in aspects was performed by Hanneman and Kiczales (2001). They developed the Aspect Mining Tool (AMT) which presents an interface that can be used with text based searches based on regular expressions. The Aspect Browser (Griswold, Yuan, & Kato, 2001) is another example of a tool that can be used when mining aspects. This tool uses a combination of the source code mapping tool Nebulous and Aspect Emacs. Aspect Emacs is an Emacs-Lisp extension to GNU Emacs which is used to provide map-indexing, map-insert, and code editing functionality. DynAMiT detects method entry and exit points and uses this information to build execution traces (Breu, & Krinke, 2004).

Aspect mining based on pattern matching and clone detection was the focus of the research performed by Bruntink, Deursen, Engelen, and Tourwe (2005). These researchers annotated manually clones in code and then used three different tools in the aspect mining effort. The goal of this research was to investigate the feasibility of using clone detection as a means for automated aspect mining. Shepherd, Pollock, and Tourwe (2005) used natural language

processing for aspect mining. Analysis of execution traces was the basis of the research performed by Tonella and Ceccato (2004). This analysis was a form of dynamic code analysis where the execution of main use cases was analyzed. When specific computational units are determined to be present in several use cases then these become candidates for aspects. The early tool oriented approaches have all in common that they are highly interactive, time and resource consuming, inefficient, and rely on extensive manual processing.

Bruntink, Deursen, Engelen, and Tourwe (2005) evaluated the suitability of using clone detection as a technique for identification of cross cutting concerns. Five specific cross cutting concerns were identified in an industrial C system and clone detection was used as a means for discovering these. The researchers concluded that cross cutting concerns implemented with similar code scattered throughout the code base was effectively identified using clone detection whereas other cross cutting concerns such as exception handling was not identifiable using this method. Shepherd, Gibson, and Pollock, (2004) developed an aspect mining tool that looks for code duplications using a dependency graph. The results are used to discover cross cutting concerns in the source code.

Shepherd, Palm, and Pollock (2005) implemented the Timna Eclipse plugin which is an automated classification tool that uses a combination of different analyses techniques for aspect mining. The application takes a program as input, identifies methods, and uses machine learning for classification of these into possible aspect mining refactoring candidates. The first step in the machine learning process is training. Training is performed by first tagging the methods in the software systems as either candidates or not candidates. Training is performed on JHotDraw (version 5.3). The machine learning testing phase is performed on JHotDraw and Tomcat (version 3.2). Timna demonstrated that better precision and recall was achieved when using a

combination of analyses than when a single analysis method such as when Fan-In analysis was used in isolation. In Fan-In analysis a method is denoted with m . The number of distinct method bodies that call each method m in the software system is calculated when Fan-In analysis is performed. This number for each method m is the Fan-In count for the method. Aspect mining using Fan-In analysis builds on the notion that methods with higher Fan-In counts is likely to represent functionality that is required across the application. Methods with higher Fan-In counts are therefore more likely to represent cutting concerns than methods with lower Fan-In counts (Martin, Deursen, & Moonen, 2004).

Clustering, a form of unsupervised machine learning, is another approach that has been used to mine for aspects in legacy code. Clustering has the advantage that little manual intervention is needed. Several clustering based techniques have been proposed for aspect mining. Cojocar and Czibula (2008) conducted a study where they evaluated the usefulness of clustering for aspect mining. The study compared the results when performing aspect mining using several different clustering techniques. They found that clustering can be useful if the clustering technique is successful in finding the optimal partition of the software system. They recommended that other clustering techniques should also be tried for aspect mining. Other early works in cluster based techniques include approaches based on Fan-In analysis (Martin, Deursen, & Moonen, 2004).

Fan-in analysis has, in addition to being used as a standalone method for aspect mining, also been used as the foundation for development of vector space models. Variations of these vector space models have been used as input for several cluster based aspect mining algorithms. Moldovan and Serban (2006) was one of the first to utilize vector space models for clustering. Partition and hierarchical clustering based algorithms have become popular choices for detection

of aspects in legacy code. Serban and Moldovan (2006a) developed a new aspect mining adaptation of the popular k-means clustering technique. These algorithms achieve higher degrees of automation than the standard k-means clustering algorithm by utilizing a heuristic for choosing the initial number and placement of centroid locations.

The formal model for partitioning based aspect mining was developed by Moldovan and Serban (2006). A theoretical foundation based on metrics for cluster based aspect mining was formed with this model. Subsequent research used this foundation as a framework when developing cluster based aspect mining clustering techniques. Serban and Moldovan (2006b) proposed in another study to use a Genetic Algorithm for aspect mining (GAM). The authors found that the kAM algorithm performed better than the GAM algorithm. Recommendations for future work included using different representation schemas for the GAM, to try different mutation schemas for improved accuracy, to use a heuristic when creating the initial population, and to make other changes to GA parameters. The Carla Laffra implementation of Dijkstra algorithm was used as the benchmark software for this case study. The authors recommended using JHotDraw for future case studies of GAM variations.

The GAM algorithm was later rejected in a comparative study of six different clustering algorithms for aspect mining (Cojocar, & Czibula, 2008). This study compared Hard K-Means, Fuzzy C-Means, Standard Genetic Algorithm, Hierarchical Agglomerative, and K-Means with heuristic based selection of initial number and location of centroids. The study was inconclusive as to which algorithm performed the best. The researchers recommended further studies to improve the clustering techniques and metrics. The four clustering algorithms kAM, hierarchical agglomerative with heuristic (HAM), partition clustering algorithm (PACO), and hierarchical agglomerative clustering (HACO) were compared by Cojocar, Czibula, & Czibula, (2009). In

this study HACO performed the best. HACO was the only algorithm where elements from one cross cutting concern did not mix with elements from other cross cutting concerns.

Recent model based aspect mining algorithms which automatically determine the optimal number of clusters have produced good clustering results. Rand McFadden and Mitropoulos (2012) achieved better results than prior studies with a careful selection of vector space models. Aspect mining data was used to calculate metrics such as diversity, measuring to which extent each cluster has different cross cutting methods from other concerns and dispersion, measuring how the cross cutting concerns are spread across the clusters. Recent unsupervised aspect mining clustering techniques provide for a high level of automation and perform well in identifying cross cutting concerns in legacy systems where AOP has not been used.

Aspect mining addresses only the first part of the problem of reverse engineering legacy software into aspect oriented systems (Mens, Kellens, & Krinke, 2008). Aspect refactoring is the second part of this problem. This is the process where the cross cutting concerns or aspects seeds are transformed into aspect modules with corresponding point cuts and advice forming an aspect oriented software system (Binkley, Ceccato, Harman, Ricca, & Tonella, 2005; Monteiro, & Fernandes, 2004). Using tools for visualization is a well-established practice within the reverse engineering community (Fabry, Kellens, & Ducasse, 2011). Tools that depend on visualization are available for examining control flow and program comprehension in software that was originally developed as aspect oriented systems or legacy systems that have already been converted to aspect oriented systems. This study will apply aspect visualization to legacy software at the aspect mining step which is before any aspect refactoring has taken place.

Visualization of Aspect Mining Results

Visualization tools exist for study of software where the crosscutting concerns have already been implemented as aspects. Understanding execution flow in applications where cross cutting concerns have been implemented as aspects can be challenging. It can be hard to understand how advice, such as before, after and around affects execution flow and at which join points these are executed for the given point-cut expressions. AspectJ, the leading Interactive Development Environment for aspect oriented programming, has visualization capabilities for aspects. Coelho and Murphy (2006) created a visualization tool which allows the software developer to browse crosscutting structures and to display these in diagrams. The diagrams show the advice, the advising methods and their relationships. *Asbro* is an aspect browsing and visualization tool which uses tree maps to show packages and classes that contain aspects (Pfeiffer, & Gurd, 2006). *Aspect Maps* was developed to show at which join points aspects execute (Fabry, Kellens, & Ducasse, 2011; Fabry, & Bergel, 2013). This tool was designed to help software developers understand the program flow in complex situations where multiple aspects intervene at the same point cut.

Little research has been conducted in order to visualize automated aspect mining results. Shepherd and Pollock (2005) created the *Aspect Miner and Viewer* (AMV) which performs aspect mining using agglomerate hierarchical clustering (ACL) to group methods that belongs to the same cluster. A simple distance measure is used to find groups of methods that belong to the same cross cutting concern. The viewer presents the clusters with the cross cutting results, the methods that belong to the clusters, and the editor pane displays the Java file for a method. The tool is limited in that it only provides for the ability to use one clustering method as input.

Manual inspection is performed to identify cross cutting concerns in the clusters. Methods are clustered based on common substrings in their names (Czibula, Cojocar, & Czibula, 2009).

Orphir, is a tool that was developed for automatic identification of aspect mining candidates (Shepherd, Gibson, & Pollock, 2004). The tool identifies cross cutting concerns that can be refactored into before advice using a four-step process: (1) Construct a source level Program Dependency Graph (PDG) to detect clones. (2) Identify a set of refactoring candidates. (3) Filter undesirable refactoring candidates. (4) Coalesce related sets of candidates into classes. Good results were reported with precision of more than 90% of the identified candidates as desirable for aspect mining refactoring. The visualization framework contains an aspect viewer that was implemented as an Eclipse plugin.

Chapter 3

Methodology

Overview of Research Methodology

The methodology and the steps that was performed in this study can best be explained by describing the design and steps performed by the aspect mining and visualization tool. This tool was built as part of this study. The aspect mining and visualization tool takes a set of input parameters, performs aspect mining on the JHotDraw source code and visualizes the cross cutting concerns identified in the source code. JHotDraw was chosen as the source code because it has a set of well-defined cross cutting concerns and is commonly accepted as the benchmark software for aspect mining. This source code has been used in many prior aspect mining studies including (Martin, Moonen, & Deursen, 2006; Rand McFadden, & Mitropoulos, 2012; Moldovan, & Serban, 2006; Shepherd, Gibson, & Pollock, 2004).

Aspect Mining and Visualization Tool

The aspect mining and visualization tool (figure 1) accepts input parameters as described below:

- (1) JHotDraw is the legacy source code that was mined for cross cutting concerns. This source code was seeded with code clones so that all categories of cross cutting concerns could be mined for.
- (2) The type of model based clustering to perform. The user chose to use, either the model-based algorithm (MCL) implemented by Fraley and Raftery (2006) in the R language (mclust package: Mclust), or the model-based agglomerative hierarchical clustering

algorithm (HC) implemented by Raftery and Dean (2006) in the R language (mclust package: hc, hclass).

- (3) The user chose to apply one of six vector space models when performing the clustering process:

(1) fanIn_numCallers, (2) fanIn_hasMethod, (3) sigTokens, (4) fanIn_sigTokens, (5) fanIn_numCallers_sigTokens, and (6) fanIn_numCallers_hasMethod_sigTokens.

- (4) One or more of 10 categories of cross cutting concerns to mine for:

(1) Ordered Method Call, (2) Code Clone, (3) Unique Class Fan In, (4) Calls In Clones, (5) Calls at Beginning and End of a Method, (6) Persistence, (7) Command, (8) Observer, (9) Decorator, and (10) Cross-cutting concerns represented as interface.

- (5) The aspect mining process started when the user clicks the *Mine for Aspects* button. The Aspect Mining Results Tab in figure 1 is revealed when the aspect mining process has completed and the results were visualized.

Aspect Mining and Visualization Tool

Aspect Mining Input Parameters
Aspect Mining Results

Benchmark software used when mining for aspects:
The JHot Draw version 4.12b

Clustering algorithm used:

☒ Model Based Algorithm

☐ Model Based Agglomerative Hierarchical

Vector Space Model to use:

☒ fanIn_numCallers

☐ sigTokens

☐ fanIn_numCallers_sigTokens

☐ fanIn_hasMethod

☐ fanIn_sigTokens

☐ fanIn_numCallers_hasMethod_sigTokens

Clone Detection Algorithm used:
A combination of a Program Dependency Graphs (PDG) and Abstract Syntax Tree (AST) is used to identify cross cutting candidates based on code clones.

Execution Trace Algorithm used:
Aspect mining using Event traces is used. This dynamic aspect mining technique was used to identify recurring execution patterns for detection of cross cutting concerns.

Cross cutting concerns mined for:

☒ Ordered Method Call

☐ Unique Class Fan In

☒ Calls At Beginning And End Of Method

☐ Command

☒ Decorator

☐ Code Clone

☒ Calls In Clones

☐ Persistence

☒ Observer

☐ Cross Cutting Concern As Interface

Mine For Aspects

Aspect mining input parameters

Figure 1: Aspect Mining and Visualization Tool User Interface.

The overall flow and sequence of steps performed by the Aspect Mining and Visualization Tool is depicted below (Figure 2). These steps were:

- (1) Aspect mining using model based clustering as was performed by (Rand McFadden, & Mitropoulos, 2012).
- (2) Aspect mining based on clone detection was performed. Two Program Dependency Graph algorithms was applied (Komondoor, & Horwitz, 2001; Krinke, 2001).
- (3) Aspect mining using event traces (Breu, & Krinke, 2004) was be performed.

(4) Aspect mining results was synthesized at step four and visualized in the source code at step five. The visualization process was guided by XML. The XML metadata was created and used to depict the individual cross cutting concerns. Each individual cross cutting concern identified in source code were annotated with the aspect mining category it belonged to.

The Aspect Mining and Visualization tool is interactive. The user can, after visualization has been completed at step five chose to run the tool again to perform a new analysis with a different set of input parameters or exit the application. A discussion of each step performed by the tool follows.

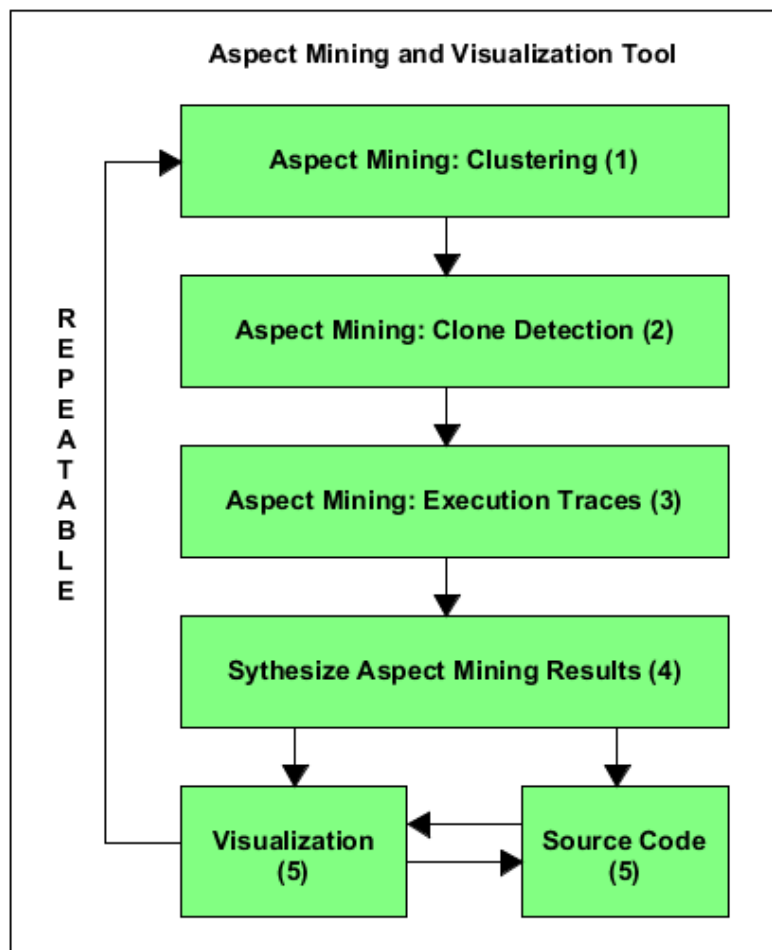


Figure 2: Aspect Mining and Visualization Tool.

Aspect Mining Using Model Based Clustering

The tool performs first model based clustering. Model based clustering was chosen because it is fully automated. This form of clustering does not require the user to specify an initial number or location of centroids prior to starting the clustering process. Rand McFadden and Mitropoulos performed model based clustering for aspect mining with good results (2012) and the methodology with corresponding steps that was performed in that study was repeated in this study.

The vector space models were in the first step constructed from raw data collected using the Eclipse FINT plugin (Martin, Deursen, & Moonen, 2004). The plugin was configured with no filtering and threshold set to zero. The resulting text file produced by the FINT tool was preprocessed to remove results from calling getters and setter and methods belonging to Java utility methods. A filter was applied so that only methods which belong to classes that are under the root package CH.* was included in the results. This ensured that the clustering algorithm was only mining methods in application source. The results were written to a file that contains application classes, method signatures, FIV values, number of calling classes, and tokenized method signatures.

The data read from this input file was further filtered and grouped before six vector space models was created. The user selects one of these vector space models when performing aspect mining (see figure 1 above). These vector space models are the same as those used by Rand McFadden and Mitropoulos (2012):

- (1) Model 1: fanIn_numCallers - A two-dimensional vector where method $M_1 = \{FIV, CC\}$.

The FIV represents the fan-in value, and the CC represents the number of calling classes.

- (2) Model 2: fanIn_hasMethod - The x -dimensional vector where method $M = \{FIV, B_1, B_2, \dots, B_n\}$ and $x = 1 + m$. The FIV represents the fan-in value and B_i represents whether the method M is called by a method in class C_j ($1 \leq j \leq m$). The m value represents all the application classes in the software system. $B_j = \{1 \text{ if } M_i \text{ is called from at least one method in class } C_j, 0 \text{ otherwise}\}$.
- (3) Model 3: sigTokens - The x -dimensional vector where method $M_i = \{O_1, O_2, \dots, O_p\}$ and $x = p$. Each method signature is split into tokens (a through z) and each unique token represents an attribute A_a through A_z . The p represents the summation of all unique tokens from all the methods ($M_i * A_{iz}$) after filtering out the duplicates and non-significant ones, such as “in”, “to”, etc. $O_j = \{1 \text{ if } M_i \text{ has attribute } A_h \text{ that equals } O_j, 0 \text{ otherwise}\}$.
- (4) Model 4: fanIn_sigTokens - Combines the FIV described in model 1 and 2 with model 3.
- (5) Model 5: fanIn_numCallers_sigTokens - Combines model 1 and model 3.
- (6) Model 6: fanIn_numCallers_hasMethod_sigTokens - Combines model 1, model 2, and model 3.

The one of the two below listed model based clustering algorithms that was selected as the input parameter for model based clustering in figure 1 above, will then be used in the clustering process:

- (1) The model-based algorithm (MCL) implemented by Fraley and Raftery, (2006) in the R language (mclust package: Mclust). This algorithm uses the EM initialized by the hierarchical clustering for the parameterized Gaussian mixture models.
- (2) The model-based agglomerative hierarchical clustering algorithm (HC) implemented by Fraley and Raftery, (2006) in the R language (mclust package: hc, hclass). This algorithm

is based on the maximum likelihood criteria for MVN mixture models parameterized by eigenvalue decomposition.

The Aspect Mining and Visualization Tool saved the result from the model based clustering so that this can later be synthesized with the aspect mining that was performed the next two steps.

Aspect Mining Using Clone Detection

Aspect mining using clone detection was the second aspect mining technique that was performed after input parameters were collected. Shepherd, Gibson, and Pollock (2004) developed a method level automated aspect mining technique based on cloning which looks for clones at the beginning of each method. A modified version of this algorithm was used. This modified version looks for code clones in the beginning, at the end, and clones that covers the entire method. This made it possible to identify cross cutting candidates for before, after and around advice. Code clones appearing in the beginning of methods are candidates for before advice. Code clones appearing at the end of methods are candidates for after advice. Code clones appearing at the beginning and the end but not in the middle are candidates for around advice. This is an expansion of the algorithm used by Shepherd, Gibson, and Pollock (2004). Their focus was on before advice and the algorithm used was therefore only concerned with clones appearing in the beginning of methods. The algorithm is performed in three steps.

The goal in the first, the *identify step*, was to create Program Dependency Graphs (PDG) (Ferrante, Ottenstein, & Warren, 1987) for all methods in the system and to identify pairs of these PDGs. These pairs represent code clones. The source level PDG is built on the IR level and points back to the source level by collapsing nodes at the same source code statement.

Comparison of the Abstract Syntax Tree (AST) at the statement level (Baxter et al. 2003) was performed to improve the accuracy of the algorithm. Comparison of AST alleviates many of the problems associated with lexical comparisons. Only control edges and not data edges between vertices are followed when constructing the individual PDGs. This improves the performance of the algorithm. This is necessary because looking for code clones in the middle and the end of methods is more time consuming than looking for code clones just in the beginning of methods.

The output from the first step was a set of paired PDGs representing code clones. Undesirable candidates PDGs pairs are filtered out by following the data dependencies between the individual vertices in the PDGs. The *Similar Data Dependence* filter discarded pairs of PDGs when the PDGs in a pair have different data dependencies. An *Outside Data Dependence* filter is performed to eliminate PDGs where the data dependence coming on to the PDGs are different because AspectJ does not allow variables referenced before the advice to be referenced in the aspect. The filters applied in this second step are the same as those applied by Shepherd, Gibson, and Pollock (2004).

The goal of the third step was to coalesce candidate sets from pairs of PDGs. The algorithm used by Baxter et al. (2003) which identifies pairs of clones was expanded to identify sets of clones. This is necessary because duplicate cross cutting concern code is frequently located in multiple modules of the source code. This was performed by taking one PDG from each pair of PDGs identified in step two and comparing these to each other. The ASTs that is created for each PDG in step one is used in this comparison. The result of coalescing the candidates is identification of groups of PDGs which are the same. Remembering that each PDG represents a code segment, the net result is that code clones in the software have been identified. These are the code clones that the code clone detection algorithm could identify.

Aspect Mining Using Event Tracing

Aspect mining by tracing events during program execution was the third form of aspect mining technique performed. Aspect mining using event trace is different than most other aspect mining techniques in that dynamic analysis is applied to identify aspect candidates. The data pool to be analyzed was generated by collecting event trace information from program executions. This study utilized the aspect mining technique that was developed by Breu and Krinkle (2004) where event traces were constructed by recording when execution flow enters and exits methods. The sequence of method entry and exit events were recorded for each use case executed in the software system. It is the patterns of events formed during method executions, the frequency at which these occur, and the context in which these occur that determines whether aspect candidates are located or not.

An example illustrates how Breu and Krinkle (2004) performed aspect mining using event traces. Figure 2 depicts sequences of event traces harvested in a fictitious program execution in a software system. In this example the method entry and exit points are recorded with one entry per line. The notation used to represent a method is `m(){ }`. Six different methods A, B, C, D, E, and F are represented. Method pair executions are of interest in the analysis of event execution traces.

Event trace information is depicted in Figure 3. Sequentially executed methods are at the same level of indentation whereas when one method calls another then the second method is at the next level of indentation. Line 1 and 2 represents the sequential execution of two methods `A(){ }` and `B(){ }`. This is called an outside relation. An inside before relation is represented on line 2 and 3. In this case method `B(){ }` calls method `C(){ }`. The return from one method to the

calling method is called an inside after relation. One such example is method D()`{ }` on line 4 returning to method B()`{ }` on line 2. These are also the call sequence relations that are of interest when looking for cross cutting concerns based on method event trace information. The example below shows a simple trace of events that have been recorded as part of fictitious program executions:

```

1      A ( ) { }
2      B ( ) { }
3          C ( ) { }
4          D ( ) {
5      }
6      E ( ) { }
7      A ( ) { }
8      B ( ) {
9          C ( ) {
10             F ( ) { }
11         }
12         D ( ) { }
13         E ( ) { }
14     }
```

Figure 3: Event Trace Information from Fictitious Program Executions.

The event traces were examined for recurring execution patterns. Recurring execution patterns in the software system represents recurring functionality in the system which may contain aspect mining candidates. Classification and analysis of execution patterns is necessary to identify potential aspect candidates.

Three conditions must to be satisfied for an execution relation to be identified as a cross cutting concern: (1) The execution relation must be *recurring*. Recurring is defined as when at least two instances of the execution relation must be present. (2) The relation must be present in two or more different *calling contexts*. A calling context is composed of the method that is calling into the execution relation and the method that is called by the method in the execution relation. (3) The execution relation must be *uniform*. Consider an inside relation where method

$A()\{\}$ is always called before method $B()\{\}$. The relation is said to be uniform if $A()\{\}$ is always called before method $B()\{\}$. Consider and inside before relation where method $C()\{\}$ is calling method $D()\{\}$. This relation is uniform if method $C()\{\}$ is always calling method $D()\{\}$.

Consider and inside after relation where method $D()\{\}$ is always returning to method $B()\{\}$. This relation is uniform if $D()\{\}$ is always returning to method $B()\{\}$. The relations that satisfies the constraints of recurring, uniform and are present in more than one calling context are aspect candidates since these represent potential cross cutting concerns. The above explanation with associated example represents an abbreviated version of the full discussion with mathematical model that is presented in Breu and Krinkle (2004).

Aspect mining using event traces consisted of three steps. These steps are the same as those Breu and Krinkle, (2004) performed in their aspect mining study. The first step was to identify the set of use cases in the legacy software that will be executed to obtain execution trace information. JHotDraw was the legacy software that was evaluated in this study. The use cases that was executed included the class(es) containing main method and the input parameter(s) that the main method(s) is called with. Focus was on use cases which are most likely to contain cross cutting concerns (Ceccato, & Tonella, 2009).

The second step consisted of collecting call stack information for all method entry and exit points during execution of each use case. This was performed by an aspect that was written in Java. This aspect is depicted in Appendix C. The aspect was configured with a point cut expression that records method entry and exit information for each method visited in the software when executing the various use cases.

The third and final step consisted of executing a program module that was written as part of the Aspect and Visualization Tool. This module analyzes the execution trace information

produced as part of step 2. The trace output was analyzed by looking for outside before, outside after, inside before, and inside after execution relations. The discovered execution relations were evaluated and a set of cross cutting concern candidates was formed from the execution relations which are *uniform*, appears in different *calling contexts*, and are *consistent*. This set of cross cutting concern candidates represents the result of aspect mining using event traces (depicted at step 3 in figure 2 on page 27).

Synthesizing Aspect Mining Results

The Aspect Mining and Visualization Tool synthesized the results after the three aspect mining algorithms has been completed. The synthetization process resulted in identification of cross cutting concerns that belongs to sets as illustrated in the below Venn diagram.

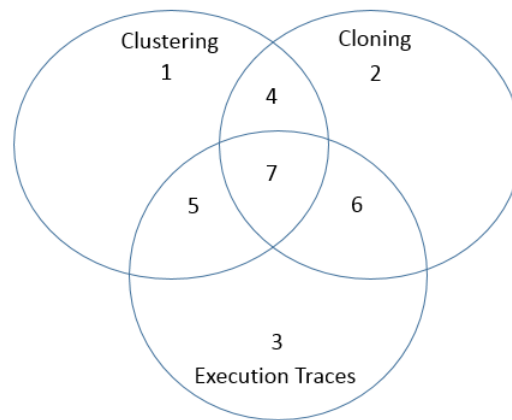


Figure 4: Venn Diagram Representing Identified Sets of Cross Cutting Concerns.

Region 1 through 3 represents cross cutting concerns identified by a single aspect mining technique. Region 4 through 6 represents cross cutting concerns identified by two aspect mining techniques. Region 7 represents cross cutting concerns identified by all three aspect mining techniques. The amount of knowledge about mined cross cutting concerns increases with the number of aspect mining techniques able to identify the cross cutting concerns. Likewise, the

ability to classify a cross cutting concern in a category increases with the amount of knowledge known for the cross cutting concern. Most knowledge is known about cross cutting concerns that are in region 7. Less knowledge is known about cross cutting concerns that are in regions 4, 5, and 6. Least knowledge is known about cross cutting concerns located in regions 1, 2, and 3. The knowledge that was extracted about the cross cutting concerns identified by the three cross cutting concern algorithms was used to categorize the various cross cutting concerns.

An example where a cross cutting concerns was identified as belonging to the *Calls at Beginning and End of a Method* category illustrates this: A cross cutting concern was identified after clustering because of its presence in a cluster. The same concern was also identified by aspect mining using execution traces. Lastly, the concern was identified by the clone detection aspect mining algorithm. The following deductions and conclusions were drawn by the Aspect Mining and Visualization Tool during the Synthesize step: (1) The two methods in question are likely to be part of cross cutting concern due to their presence in clusters. (2) The identified methods are consistently present at the beginning and end of other enclosing methods. The clone detection algorithm revealed that there were no code clones between the first and last method calls in these enclosing methods. The cross cutting concern were therefore classified as belonging to the fifth category: *Calls at Beginning and End of a Method*. The software developer can now refactor the cross cutting concern into a new aspect with corresponding around advice. Similar analysis was performed for all cross cutting concerns in the ten cross cutting concern categories.

The Aspect Mining and Visualization Tool user interface switches automatically to the *Aspect Mining Results* tab (see Figure 1) when the automated analysis had been completed. Visual representations appear for each cross cutting concern that was located.

Summary statistics is displayed at the bottom showing:

- (1) Number of cross cutting concerns located using *Clustering*.
- (2) Number of cross cutting concerns located using *Event Traces*.
- (3) Number of cross cutting concerns located using *Clone Detection*.
- (4) Number of cross cutting concerns located using *Clustering* and *Clone Detection*.
- (5) Number of cross cutting concerns located using *Clustering* and *Event Traces*.
- (6) Number of cross cutting concerns located using *Clone Detection* and *Event Traces*.
- (7) Number of cross cutting concerns located using *Clustering*, *Clone Detection*, and *Event Traces*.

Summary statistics displays also how many cross cutting concerns that was found for each of the selected categories.

Visualization of Cross Cutting Concerns in Source Code

The aspect mining and visualization tool was implemented as a state of the art RESTful Web services Web application. The UI was implemented as a thin web application using a combination of HTML, Bootstrap (Bootstrap, 2018), and Angular (AngularJS, 2018). The UI calls methods on the RESTful web application using AJAX. The client AJAX call to start mining for aspects calls the *updateAspectMine* method on the RESTful web application. Basic validation was performed on the *AspectMine* java class which is passed from the Angular client as a JSON object to the RESTful web services implementation. This web application calls the *mineForAspects* Java method on the controller jar file. The controller jar file dispatches call to three separate modules packaged in their respective jar files, one for each aspect mining technique. The Java code in these Jar files perform the aspect mining operations. The results from these mining operations were returned to the controller jar file which synthesized the results and returned these to the RESTful web application. The results were converted from Java class to JSON object representation and displayed in the user interface in the web application for the user to see. If the user chose to drill down into individual concerns for additional information, then this information was retrieved and presented in the UI.

Fabry, Kellens, and Ducasse (2011) have set forth six guidelines that should be fulfilled for a visualization tool to perform well. These guidelines, listed next, was followed when implementing the visualization tool: (1) The user of the visualization tool should not be overwhelmed by the number of colors used. (2) The information conveyed should not be too complex and it should not be too simplistic as this would leave out important detail. (3) There should be a clear mapping between the entities presented and the domain being visualized. (4) The visualization should provide the right density of information. (5) The visualization should scale well and depict adequate detail when viewing small samples and when viewing large quantities of data. (6) The visualization should facilitate good interactivity and enable ease of use.

Resource Requirements

The proposed research was implemented using Java version 1.8 with the IntelliJ Interactive Development Environment (IntelliJ IDE, 2018). All visualization components were implemented using the AngularJS framework (AngularJS, 2018) with Bootstrap for CSS (Bootstrap, 2018) for rendering results in the UI. JSON objects were used for transportation of data between the UI web application and the RESTful web application. A 64-bit version of System R installed on Windows 7 was used for the clustering process.

The study was performed using the JHotDraw (version 5.4b1) benchmark application. JHotDraw has been used extensively in prior aspect mining research. A 64-bit Dell windows laptop computer with 64 GB of memory and an Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz processor was used for all Java development and documentation work. This research

study did not involve human subjects and no approvals was therefore needed from the Institute Review Board (IRB).

Chapter 4

Results

Three aspect mining techniques was used when mining for aspects: (1) Code Clone Detection, (2) Execution Traces, and (3) Cluster Analysis.

Data Generation

Aspect Mining: Code Clone Detection

The amv-clone-detection.jar file contains the code that was written to perform aspect mining using code clone detection. This analysis found cross cutting concerns within the cross cutting concern category of *Code Clone*. Specifically, the three different types: (1) Before Advice, (2) After Advice, and (3) Around Advice. Duplicate methods were also detected. These are candidates for code refactoring that removes duplicates. This aspect mining process was performed in two stages:

Stage I: Create a collection of method representations for each method in the entire JHotDraw source code. Stage II: Compare the method representations in the collection of method representations identified in stage I with each other to identify code clones. The steps at each stage is explained in detail:

Steps at Stage I:

(1) Identify all source files in the source file directory to be scanned. A FileScanner class was developed to accomplish this task. The FileScanner class recursively scans and identifies all Java files in the entire source tree to be examined. The FileScanner class takes three input parameters: (a) The source root directory from where to start the file scanning. (b) A list of directories which should be omitted from the file scanning process. (c) The file extension of the

files to be scanned. The source root directory to start scanning was *CH/ifa* which is the top package in JHotDraw source directory. The root packages *samples*, *jdk11*, *jdk12* are also part of the JHotDraw source distribution, but contains testing related files and were therefore listed as part of directories to skip in the file scanning process. Only files with file extension *java* was included in the file scan and this extension was therefore passed in as the third parameter for the extension of files to be included in the file scanning process. The `FileScanner` class returns an `ArrayList` of files for further processing. (2) Each Java file is examined to identify the methods in each of these Java files. This was done by building an Abstract Syntax Tree for each class contained in the java source files. The widely used open source Java Parser and Aspect Syntax Tree (Van Bruggen, 2017) project was used for this purpose. (3) A collection of abstract syntax tree representations of the methods in each class was extracted. All `MethodRepresentation` class instances for all Java files were stored in an `ArrayList`. This `ArrayList` represents all methods in the JHotDraw source distribution and is the input to the next stage in the aspect mining process.

Steps at Stage II:

(1) All `MethodRepresentations` instances were compared with each other to find code duplications. There are five possible outcomes when any two `MethodRepresentations` are compared with each other. (1) No code duplication, (2) code duplication in the beginning of the two compared methods, (3) code duplication in the middle of the methods compared, (4) code duplication at the end of the methods, and (5) the methods compared are fully duplicated methods. Code duplication at the beginning of compared methods are cross cutting candidates for Before Advice. Code duplication in the middle of methods are candidates for Around Advice. Code clones for methods at the end of methods are candidates for After Advice.

Candidates from all categories were found in the code.

Naturally, most code clone comparisons resulted in that the methods compared were found to be totally different from each other. This means that there was no code duplication between the methods compared. Thirty-three instances were found where methods were complete clones. Forty methods were found to have the same beginning. One method was found to have the same beginning and ending. Five methods were found to have the same ending. A code token is delimited by a semicolon on both ends. Code comments were excluded before performing method clone comparisons. Tabulated results are located below in the Data Analysis section. Fully duplicated code of methods in different classes was also found:

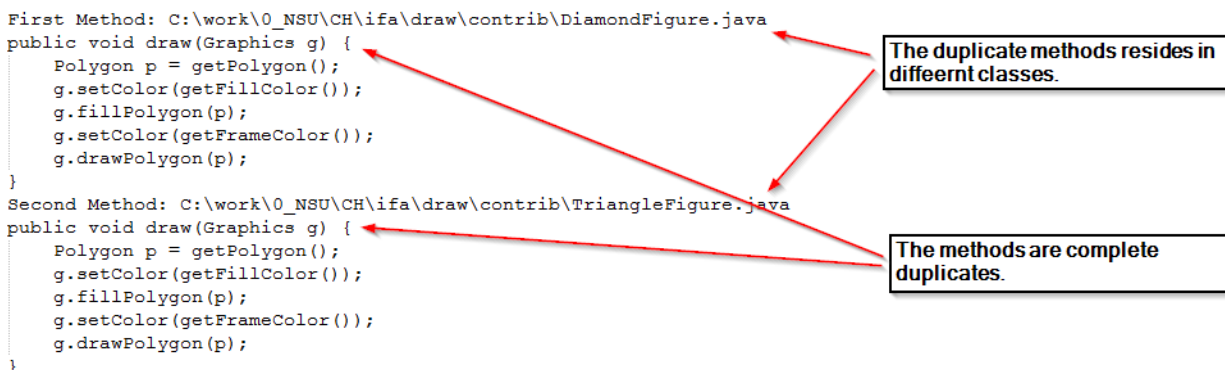


Figure 5: Fully Duplicated Code for Methods in Different Classes.

As mentioned, code duplication at the beginning of compared methods are cross cutting candidates for Before Advice. See example below:

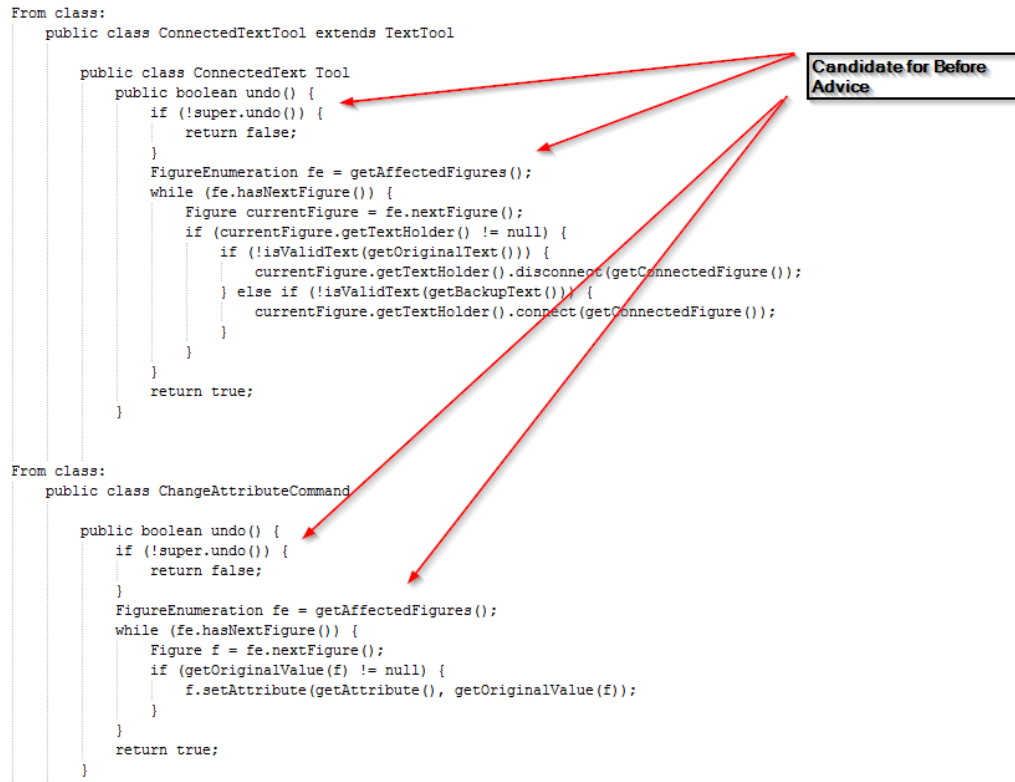


Figure 6: Duplicated Code at the Beginning of Methods, Candidate for Before Advice.

Code duplication at the end of compared methods are cross cutting concern candidates for After Advice. See example below:

```

First Method: TextAreaFigure.updateLocation
protected void updateLocation() {
    if (fLocator != null) {
        Point p = fLocator.locate(fObservedFigure);
        p.x -= size().width / 2 + fDisplayBox.x;
        p.y -= size().height / 2 + fDisplayBox.y;
        if (p.x != 0 || p.y != 0) {
            willChange();
            basicMoveBy(p.x, p.y);
            changed();
        }
    }
}

Second Method: TextFigure.updateLocation
protected void updateLocation() {
    if (getLocator() != null) {
        Point p = getLocator().locate(getObservedFigure());
        p.x -= size().width / 2 + fOriginX;
        p.y -= size().height / 2 + fOriginX;
        if (p.x != 0 || p.y != 0) {
            willChange();
            basicMoveBy(p.x, p.y);
            changed();
        }
    }
}

```

The diagram illustrates two methods, `TextAreaFigure.updateLocation` and `TextFigure.updateLocation`, which contain duplicated code at their ends. A box labeled "Candidate for After Advice" has red arrows pointing to the end of each method, indicating the location where advice can be inserted.

Figure 7: Duplicated Code at the End of Methods, Candidate for After Advice.

Only one method was found to have the same beginning and ending. The figure below shows these methods. The two methods are identical except for the two lines:

```

chunked = startPos;
chunked = startPos + 1;

```

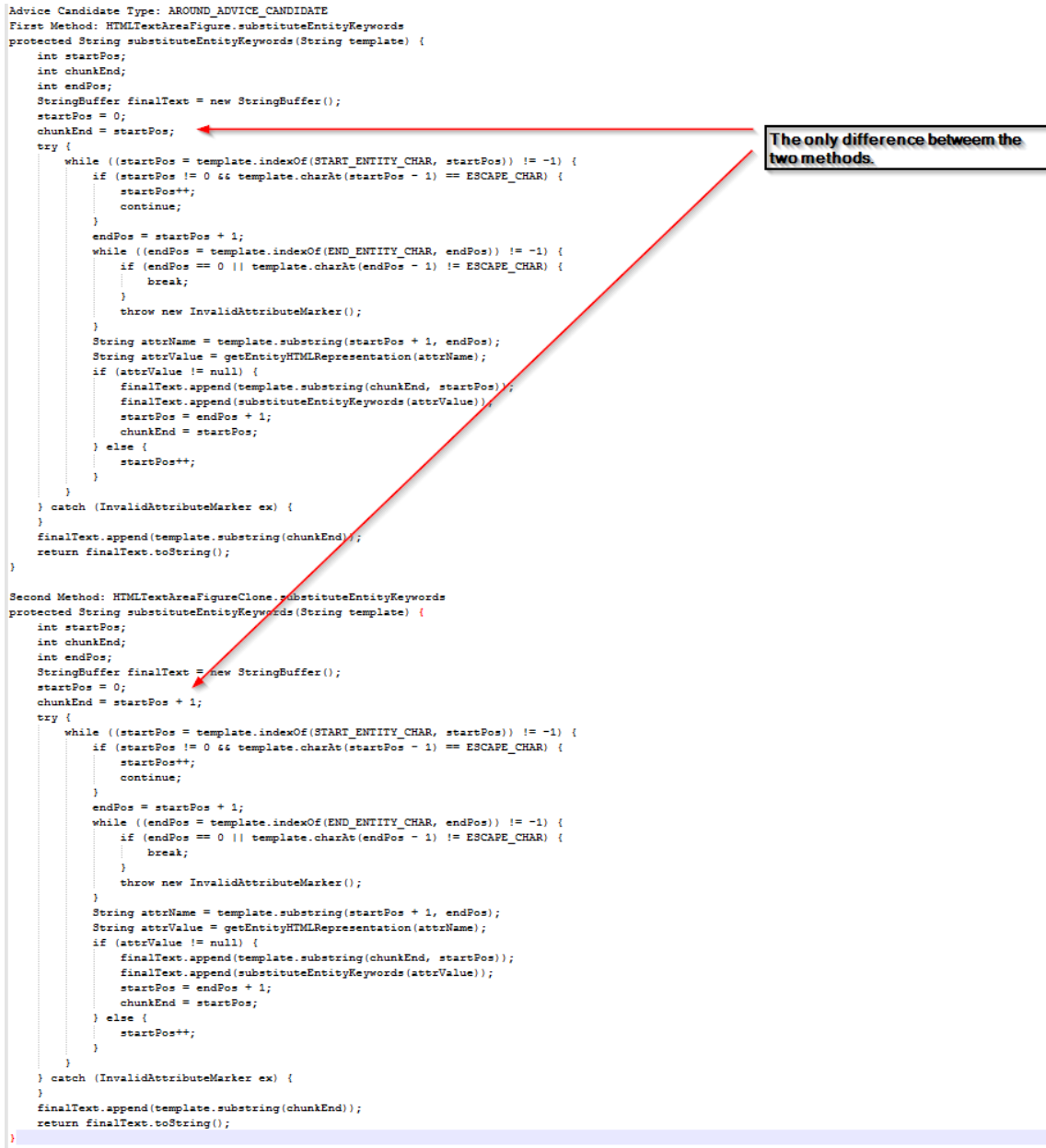



Figure 8: Method Candidate for Around Advice.

Aspect Mining: Execution Traces

The aspect mining method for execution of traces was implemented using Aspect Oriented Programming. The AMV Execution Tracing project (*amv-execution-tracing*) was developed. This project is a multimodule Maven project housing two modules, *trace-criteria*,

and *trace-subject*. These are packaged in the *trace-criteria.jar* and trace subject *trace-subject.jar* jar files respectively.

The trace criteria jar file contains only one class which is the *MethodExecutionTraceAspect* class. This class contains (a) a pointcut expression which selects the pointcuts where the advice is applied, (b) the before advice that is applied at the selected join points, and (c) the after advice which is applied after execution of the selected pointcuts. The pointcuts selected by the pointcut expression are all methods in the JHotDraw software. The before advice logs a trace of the signature. The Log4j logging framework was used to create a trace of methods called during program execution. The NDC object, which is part of the Log4J framework was used to produce indentation levels for the logged methods, and thereby show method depth in the method call trace. Before advice pushes spaces using the NDC on to the stack when entering methods, and after advice pops these off the stack with the NDC when exiting from methods. This creates an indented method calls trace log. This helps understanding the method call sequence and method call depth. The output from the execution trace run was logged to the *C:/log/method_trace.log* log file. The *MethodExecutionTraceAspect* implementation is like the one developed by Laddad, (2003). The aspect and excerpt from trace log file are depicted in Appendix C.

The trace subject jar-file contains a compiled version of all the Java files in the source code that is to be traced. In this case the entire JHotDraw software tree. In addition, the AspectJ Maven Plugin (*aspectj-maven-plugin*) was used for aspect weaving purposes. This Maven plugin wove the source code in together with the aspect criteria and produced an executable jar file *trace-subject.jar*. This was run to create the log trace. The executable jar file (*trace-subject.jar*) was created using the Maven Shade Plugin (*maven-shade-plugin*). This implementation was

inspired by Laddad, (2003). Significant effort was spent getting the Maven configuration set up correctly to get the pointcut expression, before and after advice, and to get all the proper jar files included in the build so that the desired execution trace would be logged to the Log4J log file. Further detail on the tracing aspect and the shade plugin configuration is included in Appendix C.

The JHotDraw source code distribution comes with a full set of JUnit tests covering the JHotDraw source code. These were used to generate the initial raw log of execution relations. The event trace log file produced when running these JUnit tests contains (1) relations when the JUnit test methods calls into the JHotDraw source code, and (2) relations generated when methods in the source code calls other methods in the source code. The relations extraction process consists of three steps: (1) Identify all calling relations in the trace log file. Any relations generated by calls from JUnit methods were filtered out in this step. Each calling sequence would always consist of JUnit calling in to the source code to be executed for the use case. All subsequent calling relations after the JUnit initial test method call were included. The set of all calling relations were collected and saved in a Java ArrayList. (2) Only recurring relations were considered in the next step. (3) Relations were extracted when these occurred in at least two different calling contexts. Of these, only uniform relations were included (i.e. relations in which the calling sequence is always the same). These are scattered cross cutting concerns of type *Ordered Method Call*. These can be refactored into aspects. All relations where JUnit calls were present were removed.

The automated analysis of the method tracing log file revealed a total of five different inside relation cross cutting concerns. Four of these were called from two different calling contexts and one was called from three different calling contexts. The inside relation that was

called from three different calling contexts is depicted below. All inside relations that was identified has been depicted in Appendix D.

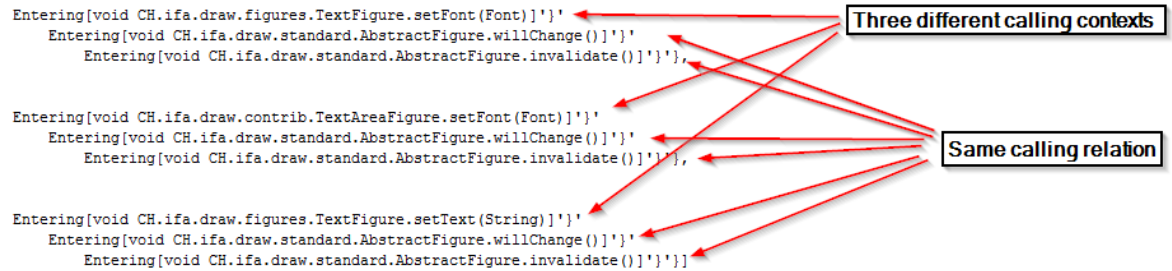


Figure 9: Identification of Ordered Method Call Cross Cutting Concerns with Inside Relation.

The automated analysis of the method tracing log file revealed a total of eleven different outside relation cross cutting concerns. Five of these were called from two different calling contexts, four were called from three different calling contexts, one was called from four different calling contexts, and one was called from six different calling contexts.

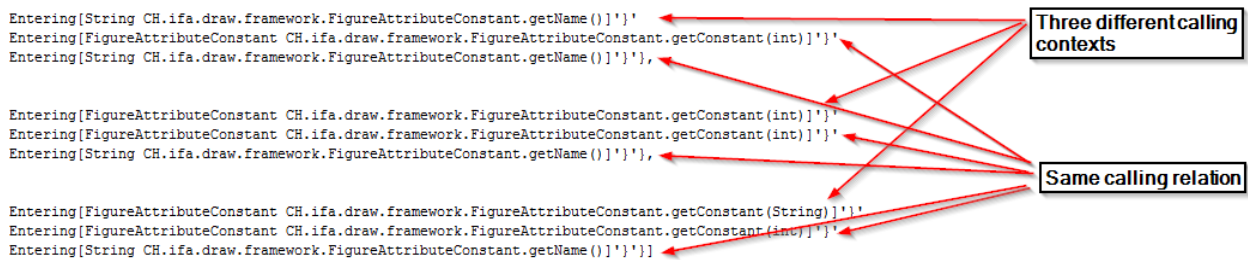


Figure 10: Identification of Ordered Method Call Cross Cutting Concerns with Outside Relation.

Please see the section on Aspect Mining using Execution Traces for further information on the characteristics of Inside and Outside Relations, the difference between these, and how these manifests themselves in source code.

Aspect Mining: Fan-in Analysis

Aspect mining based on the results of fan-in analysis was performed by the code in the `amv-fan-in-analysis.jar` module. The aspect mining using fan-In analysis takes the file `AnalysisResults.txt` as an input parameter when mining for cross cutting concerns. This file was

generated by the FINT tool with parameter settings as described at the Software Engineering Group Web Site for Fan-in Analysis Results web site

(<http://swerl.tudelft.nl/bin/view/AMR/FanInAnalysisResults>). The threshold was set at 10 so to filter out methods with too low fan-in value. The following filters were applied:

- (1) Filter out any methods from the *test* package
- (2) Filter out any utility methods:
 - CH.ifa.draw.test.* (also filtered from the set of callers - see filters for callers in FINT)
 - CH.ifa.draw.util.collections.* (this includes CH.ifa.draw.util.collections.jdk11.* and CH.ifa.draw.util.collections.jdk1.2.*)
 - CH.ifa.draw.util.CollectionsFactory
 - CH.ifa.draw.util.ReverseListEnumerator
 - CH.ifa.draw.standard.ReverseFigureEnumerator
 - CH.ifa.draw.standard.HandleAndEnumerator
 - CH.ifa.draw.standard.SingleFigureEnumerator
 - CH.ifa.draw.standard.FigureAndEnumerator
 - CH.ifa.draw.standard.HandleEnumerator
 - CH.ifa.draw.standard.FigureEnumerator
 - CH.ifa.draw.framework.FigureEnumeration
 - CH.ifa.draw.framework.HandleEnumeration

The generated results file containing only cross cutting concern seeds was 3222 lines long and contained methods that had fan-in value equal to or more than 10. A small sample of this file is depicted in Appendix D. This file was read by the fan-in analysis module in the *amv-fan-in-analysis.jar* file. The concern seed methods with corresponding calling methods were loaded by the Java module into instances of the *FanInConcernSeed* class. The entire array of *FanInConcernSeed* instances was passed back to the *clone-detection.jar* file where these concern seeds were synthesized with results from aspect mining using code clone detection and execution traces (see Figure 11 for details). The synthesized results for the different cross cutting concern categories were passed back to the web service interface and presented to the user.

Aspect Mining: Cluster Analysis

Aspect mining using cluster analysis was found to be effective for cross cutting concerns of the following categories: (1) Decorator, (2) Observer, (3) Command, and (4) Persistence. The method call patterns was most similar for these types of cross cutting concern categories. Consequently, the implementing methods for the same concern were most likely to appear in the same cluster. Cluster analysis brings the additional capability that it can identify sets of methods that belongs to a specific concern. Cluster analysis was for this reason found to be effective in identifying complex cross cutting concerns such as the above mentioned which are based on design patterns.

Capturing Aspect Mining Visualization Metadata in XML

The cross cutting concerns found for the various cross cutting concern categories were represented in XML. This was done so that the aspect visualization user interface implementation has a standard format for representing the retrieved cross cutting concern information in the user interface. The XML representation accounts for two different types of cross cutting concern categories: (1) The most common is that where there is one method that is called by other methods. This requires a one to many method caller to method called XML representation. (2) Another type is that when there is a collection of methods that composes a cross cutting concern. This is the case for cross cutting concerns that are based on code clones. This requires a different form of XML representation. All cross cutting concern categories were successfully represented using either one of these two types of XML representations.

An aspect mining session produces an aspect mining result XML. The aspect mining result XML contains XML representations of all cross cutting concerns that were found. These

are grouped by cross cutting concern category. The below example XML in figure 11 illustrates this. In this case the aspect mining that was performed resulted in that two cross cutting concerns were found. One cross cutting concern type of *Duplicate Method* and another cross cutting concern of type *Calls At The End Of A Method*. The general DTD representing valid XML results is included in Appendix E.

```
<ASPECT-MINING-RESULT>
  <CROSS-CUTTING-CONCERN-CATEGORIES>
    <CROSS-CUTTING-CONCERN-CATEGORY>
      <CROSS-CUTTING-CONCERN-CATEGORY-TYPE>CODE_CLONE_BEFORE_ADVICE</CROSS-CUTTING-CONCERN-CATEGORY-TYPE>
      <DUPLICATE-METHOD>
        <METHOD-NAME>undo</METHOD-NAME>
        <FILE-NAME>C:\work\0_NSU\CH\ifa\draw\figures\ConnectedTextTool.java</FILE-NAME>
        <STARTING-LINE-NUMBER>173</STARTING-LINE-NUMBER>
        <ENDING-LINE-NUMBER>187</ENDING-LINE-NUMBER>
      </DUPLICATE-METHOD>
      <DUPLICATE-METHOD>
        <METHOD-NAME>redo</METHOD-NAME>
        <FILE-NAME>C:\work\0_NSU\CH\ifa\draw\figures\ConnectedTextTool.java</FILE-NAME>
        <STARTING-LINE-NUMBER>130</STARTING-LINE-NUMBER>
        <ENDING-LINE-NUMBER>151</ENDING-LINE-NUMBER>
      </DUPLICATE-METHOD>
    </CROSS-CUTTING-CONCERN-CATEGORY>
    <CROSS-CUTTING-CONCERN-CATEGORY>
      <CROSS-CUTTING-CONCERN-CATEGORY-TYPE>CALLS_AT_THE_END_OF_A_METHOD</CROSS-CUTTING-CONCERN-CATEGORY-TYPE>
      <REFERENCED-METHOD>
        <METHOD-NAME>readStorable</METHOD-NAME>
        <FILE-NAME> C:\work\0_NSU\CH\ifa\draw\util\StorableInput.java</FILE-NAME>
        <STARTING-LINE-NUMBER>10</STARTING-LINE-NUMBER>
        <ENDING-LINE-NUMBER>87</ENDING-LINE-NUMBER>
      </REFERENCED-METHOD>
      <REFERENCING-METHODS>
        <REFERENCING-METHOD>
          <METHOD-NAME>read</METHOD-NAME>
          <FILE-NAME>C:\work\0_NSU\CH\ifa\draw\contrib\GraphicalCompositeFigure.java</FILE-NAME>
          <STARTING-LINE-NUMBER>338</STARTING-LINE-NUMBER>
          <ENDING-LINE-NUMBER>342</ENDING-LINE-NUMBER>
        </REFERENCING-METHOD>
        <REFERENCING-METHOD>
          <METHOD-NAME>read</METHOD-NAME>
          <FILE-NAME>C:\work\0_NSU\CH\ifa\draw\contrib\html\TextHolderContentProducer.java</FILE-NAME>
          <STARTING-LINE-NUMBER>90</STARTING-LINE-NUMBER>
          <ENDING-LINE-NUMBER>95</ENDING-LINE-NUMBER>
        </REFERENCING-METHOD>
        <REFERENCING-METHOD>
          <METHOD-NAME>read</METHOD-NAME>
          <FILE-NAME>C:\work\0_NSU\CH\ifa\draw\standard\LocationConnector.java</FILE-NAME>
          <STARTING-LINE-NUMBER>96</STARTING-LINE-NUMBER>
          <ENDING-LINE-NUMBER>99</ENDING-LINE-NUMBER>
        </REFERENCING-METHOD>
      </REFERENCING-METHODS>
    </CROSS-CUTTING-CONCERN-CATEGORY>
  </CROSS-CUTTING-CONCERN-CATEGORIES>
</ASPECT-MINING-RESULT>
```

Figure 11: Sample XML Representation of Aspect Mining Results.

Data Analysis

Data analysis of cross cutting concern categories identified by the Aspect Mining and Visualization Tool follows. The table below shows the results that was found for cross cutting category of *Ordered Method Call*. The aspect mining technique used to identify cross cutting concerns belonging to this cross cutting category was *Execution Traces*.

Type of Relation	Number of Different Calling Contexts	Number of Instances
Inside Relation	2	4
Inside Relation	3	1
	Total Number of Inside Relations	5
Outside Relation	2	5
Outside Relation	3	4
Outside Relation	4	1
Outside Relation	6	1
	Total Number of Outside Relations	11
	Total Number of Relations (Inside and Outside)	16

Table 1: Ordered Method Call Analysis Results.

The Aspect Mining and Visualization Tool reported results as follows when mining for aspects in the category of code cloning detection. Thirty-three methods were found to be complete clones of each other. These are candidates for reverse engineering using *Extract Method* refactoring (Fowler, 1999).

Two configurable parameters were used to determine how many tokens in each method would have to be the same for methods to contain code clones either at the beginning of a method or at the end of a method. The top of method token threshold and bottom method threshold parameters were both set to 4. That means that at least 4 of the tokens in the beginning or ending would have to be the same for one method to be reported to contain a clone of another method.

Forty methods were found to have cloned code at the beginning of the methods. Five methods were found to have cloned code at the end of the methods. Two methods were found to have code clone at the beginning and code clone at the end of the method. Methods with code clones at the beginning are candidates for before advice aspects. Methods with code clones at the

end are candidates for after advice candidates. Methods with code clones at the beginning and the end are candidates for around advice. The results when mining for code clones are summarized in the table below.

Methods with beginning code clones	Methods with ending code clones	Methods with beginning and ending code clones	Methods that are complete clones of each other	Methods where no code clones were found
Before Advice Candidates	After Advice Candidate	Around Advice Candidate	Refactor using Extract Method	Methods are totally different
40	5	1	33	3,755,091

Table 2: Code Clone Analysis Results.

The cross cutting concern category of *Unique Class Fan In* is when methods from unique classes make calls to one specific method (Martin, Deursen, & Moonen, 2004). Finding cross cutting concerns by using fan in analysis is one that is particularly well suited for automation and was therefore a natural fit for this study. The procedure for generating the input file to be processed for finding the cross cutting concerns for this category is described above with the FINT tool. The results after having processed the generated input file were presented in the user interface of the Aspect Mining and Visualization tool. The calling methods are not included in the table below since these are so numerous. The called method becomes candidate for advice implementation and the calling methods become candidates for point cut locations.

Method	Fan-In Value
CH.ifa.draw.standard.AbstractFigure.displayBox()	90
CH.ifa.draw.standard.AbstractFigure.displayBox()	90
CH.ifa.draw.figures.TextFigure.displayBox()	60
CH.ifa.draw.contrib.TextAreaFigure.displayBox()	60
CH.ifa.draw.samples.net.NodeFigure.displayBox()	60
CH.ifa.draw.figures.RoundRectangleFigure.displayBox()	59

Table 3: Methods JHotDraw with the highest Fan-In values.

The cross cutting concern category of *Cross Cutting Concern as Interface* is present when code duplication is found in interface implemented methods. The duplicated code is in these instances good candidates for aspect implementation. The automated analysis starts by finding all code clones in the entire source tree. The interfaces for classes containing methods with code clones are then identified. Code clones within methods that implement the same interfaces are identified as belonging to the cross cutting concern category of *Cross Cutting Concern as Interface*. There are no natural occurrences of *Cross Cutting Concern as Interface* in the JHotDraw source code. Methods were therefore seeded with a common interface named *CommonInterface* before aspect mining for this concern were performed. The tool found all occurrences of the cross cutting concern category of *Cross Cutting Concern as Interface*. The table below table show the mining results.

Methods with beginning code clones	Methods with ending code clones	Methods with beginning and ending code clones	Code clones with common interfaces
Before Advice Candidates	After Advice Candidate	Around Advice Candidate	Refactored using Extract Method
3	2	1	15

Table 4: Cross Cutting Concern Implementing Interface Analysis Results.

The following is one such representation where these are implementing methods of the *CommonInterface* interface. These are a subset of the code clones at the beginning of a method. These are good candidates for *Before Advice* aspect implementations since the developer has already determined that these methods have similar meaning by making these implementations of an interface. *Cross Cutting Concern as Interface* also comes as candidates for *Around Advice* and *After Advice* when the code clones appear in the middle or at the end of methods.



Figure 12: Identification of Cross Cutting Concern as Interface.

The cross cutting concern category of *Calls in Clones* is present when the same method calls are made from various clones. This concern is characterized by code clones scattered in the application. The cloned code contains frequent calls to a method M which is a good candidate for a cross cutting concern that can be aspectized (Bruntink, Deursen, Engelen, & Tourwe, 2005). Mining for these cross cutting concerns was performed in four steps. (1) The code clones that had been previously identified was used as the starting point. (2) The Aspect Syntax Tree (Van Bruggen, 2017) was utilized when isolating the methods that were called from the code clones. (3) Representations of all methods in the entire JHotDraw source code were created. (4) The calling methods from the code clones were compared with all method representations and matches were saved and reported on. Table 4 below shows the identified Calls in Clones cross cutting concerns with class in which it was found, the specific method in the class, and the frequency at which these were detected. The best candidates for refactoring into cross cutting concerns are those with the highest frequency.

Class	Method	Frequency
CH\ifa\draw\util\PaletteButton.java	PaletteButton.select	1
CH\ifa\draw\contrib\zoom\DoubleBufferImage.java	DoubleBufferImage.getSource	1
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.fillPolygon	1
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.drawPolygon	1
CH\ifa\draw\contrib\html\DisposableResourceManagerFactory.java	DisposableResourceManagerFactory.createStandardHolder	1
CH\ifa\draw\util\StorableInput.java	StorableInput.readStorable	1
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.clearRect	2
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.fillRect	2
CH\ifa\draw\util\StorableOutput.java	StorableOutput.writeBoolean	2
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.getClip	5
CH\ifa\draw\util\StorableOutput.java	StorableOutput.writeInt	4
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.setColor	9
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.drawOval	9
CH\ifa\draw\contrib\zoom\ScalingGraphics.java	ScalingGraphics.fillOval	11

Table 5: Identified Calls in Clones Cross Cutting Concern.

The figure below shows a frequently recurring set of methods that are called in clones and are therefore excellent candidates for aspect refactoring.

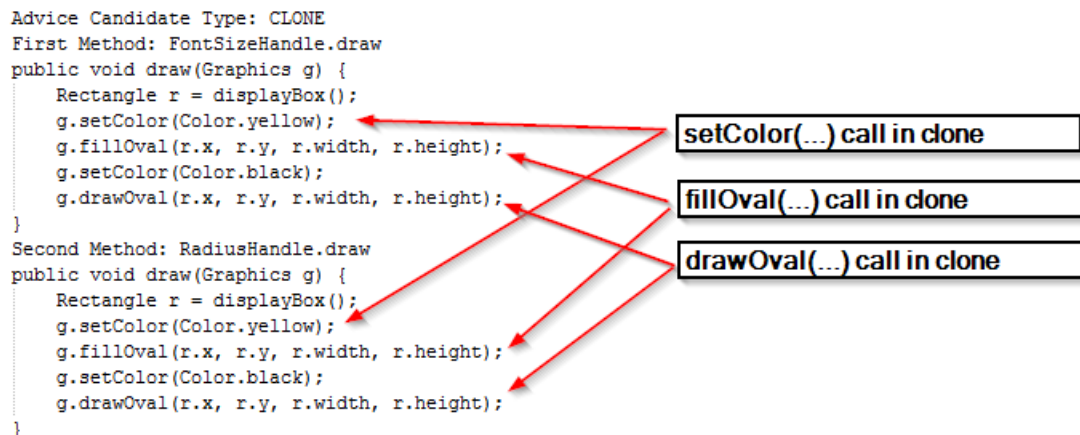


Figure 13: Example of Frequent *Calls in Clones* Cross Cutting Concern.

The cross cutting concern category of *Calls at the Beginning and End of Method* is present when the same method is called either in the beginning or at the end of methods. These are often good candidates for cross cutting concerns that can be aspectized. Examples of such are logging and methods implementing transaction demarcations such as begin transaction, commit,

and rollback. Mining for these concerns were performed in two steps. First, mine for *Calls at the Beginning of Methods* and then for *Calls at the End of Methods*.

Mining for *Calls at the Beginning of Methods* was performed in four steps. (1) Representations for all methods for all classes in the JHotDraw was first created. (2) Candidates for this cross cutting concern category was located by iterating over all these methods and extracting those which are calling another method in the first line of the method. Seventy-four such methods were discovered. (3) The Aspect Syntax Tree was then examined to identify methods which are calling out to another method in the first line of the method. (4) Methods that are calling the same method in the first line of the code were then grouped together. The methods which are calling other methods are candidates for cross cutting concerns. The table below shows the findings for cross cutting concern category of *Calls at the Beginning of Methods*. A total of nine sets of method calls at the beginning calling to a common method was found.

Class Called	Method Called	Method Called From Count
CH\ifa\draw\util\CollectionsFactory.java	CollectionsFactory.createList	23
CH\ifa\draw\contrib\zoom\DoubleBufferImage.java	DoubleBufferImage.getSource	7
CH\ifa\draw\util\FloatingTextField.java	FloatingTextField.addActionListener	2
CH\ifa\draw\contrib\ComponentFigure.java	ComponentFigure.getComponent	2
CH\ifa\draw\figures\PolyLineHandle.java	PolyLineHandle.getPointIndex	2
CH\ifa\draw\contrib\PolygonFigure.java	PolygonFigure.smoothPoints	2
CH\ifa\draw\util\StorableInput.java	StorableInput.readString	2
CH\ifa\draw\util\Bounds.java	Bounds.getLesserX	2
CH\ifa\draw\application\DrawApplication.java	DrawApplication.print	6

Table 6: Identified Calls at the Beginning of Methods Cross Cutting Concern.

The first entry in table 6 shows that 23 methods calling the `CollectionsFactory.createList` method. These calls are listed in the table below:

Method number	Methods calling the CollectionsFactory.createList method
1	<code>QuadTree.getAllWithin</code>
2	<code>StandardDrawingView.addForeground</code>
3	<code>PolygonFigure.points</code>
4	<code>RoundRectangleFigure.handles</code>
5	<code>PolygonFigure.handles</code>
6	<code>StandardDrawingView.add</code>
7	<code>RectangleFigure.handles</code>
8	<code>ImageFigure.handles</code>
9	<code>StandardDrawingView.addBackground</code>
10	<code>EllipseFigure.handles</code>
11	<code>StandardDrawingView.selectionZOrdered</code>
12	<code>DesktopEventService.getDrawingViews</code>
13	<code>UndoableAdapter.rememberFigures</code>
14	<code>GroupFigure.handles</code>
15	<code>StandardDrawing.handles</code>
16	<code>TextFigure.handles</code>
17	<code>StandardDrawing.addDrawingChangeListener</code>
18	<code>GraphicalCompositeFigure.handles</code>
19	<code>HandleEnumerator.toList</code>
20	<code>TextAreaFigure.handles</code>
21	<code>AbstractFigure.decompose</code>
22	<code>CompositeFigure.figures</code>
23	<code>ComponentFigure.handles</code>

Table 7: List of methods calling `CollectionsFactory.createList` method at the beginning.

Mining for *Calls at the End of Methods* was performed in the same four steps as described above when mining for *Calls at the Beginning of Methods*. The only difference was that instead of finding methods that had a call to another method as the first method statement, the algorithm was looking for methods that had calls to another method as the last statement in the method. More candidates were found for this category of *Calls at the End of Methods*. A total of 55 instances were found. An example of one of these with associated source code is depicted below.



Figure 14: Example of Calls at the End of Methods Cross Cutting Concern.

Drill Down and Inspect Individual Cross Cutting Concerns

The categories of cross cutting concerns that will be mined for depends on the selections that has been made by the user in the aspect mining tool. For example, when the user makes the selection for categories to mine for as depicted in figure 15, then the aspect mining results depicted in figure 16 appears after the aspect mining has been completed.

Cross cutting concerns mined for:

<input checked="" type="checkbox"/> Ordered Method Call	<input type="checkbox"/> Code Clone
<input type="checkbox"/> Unique Class Fan In	<input checked="" type="checkbox"/> Calls In Clones
<input checked="" type="checkbox"/> Calls At Beginning And End Of Method	<input type="checkbox"/> Persistence
<input type="checkbox"/> Command	<input checked="" type="checkbox"/> Observer
<input checked="" type="checkbox"/> Decorator	<input type="checkbox"/> Cross Cutting Concern As Interface

Figure 15: User Selects Cross Cutting Concerns Categories to Mine for.

Aspect Mining and Visualization Tool

[Aspect Mining Input Parameters](#)
[Aspect Mining Results](#)

Aspect Mining Result: Success

Cross Cutting Concern Category	Number Of Cross Cutting Concerns Found	Details
Calls At The Beginning Of A Method	33	Cross Cutting Concern Details
Calls At The End Of A Method	55	Cross Cutting Concern Details
Ordered Method Call, Inside Relation	40	Cross Cutting Concern Details
Ordered Method Call, Outside Relation	11	Cross Cutting Concern Details
Clone, Before Advice	40	Cross Cutting Concern Details
Clone, After Advice	5	Cross Cutting Concern Details
Clone, Around Advice	1	Cross Cutting Concern Details
Clone, Complete Clone	33	Cross Cutting Concern Details
Cross Cutting Concern As Interface	33	Cross Cutting Concern Details
Calls In Clones	33	Cross Cutting Concern Details
Cross Cutting Concern As Interface, Before Advice	3	Cross Cutting Concern Details
Cross Cutting Concern As Interface, After Advice	2	Cross Cutting Concern Details
Cross Cutting Concern As Interface, Around Advice	1	Cross Cutting Concern Details
Total	290	

Figure 16: User Clicks Cross Cutting Concern Details Link to get more Information.

The user can now click on the *Cross Cutting Concern Details* for a cross cutting concern category to drill down into and inspect individual cross cutting concerns. The user clicks to see details for *Calls At The End Of A Method* in this example (figure 16).

The below depicted section of the aspect mining tool is displayed after the user has clicked the *Cross Cutting Concern Detail* link. In this case the user clicked the link to see more details for the *Calls At The End Of A Method* cross cutting concern category. Further, the user has clicked the *CH\ifa\draw\util\StorableInput.java : readStorable* cross cutting concern. Details for the called method and the three calling methods are now displayed on the right-hand side on the screen (figure 17).

Calls At The End Of A Method	
Called Method Summary	Called / Calling Methods Detail
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : drawPolyline	Called Method - CH\ifa\draw\util\StorableInput.java : readStorable <pre> public Storable readStorable() throws IOException { Storable storable; String s = readString(); if (s.equals("NULL")) { return null; } if (s.equals("REF")) { int ref = readInt(); return (Storable) retrieve(ref); } storable = (Storable) makeInstance(s); map(storable); storable.read(this); return storable; } </pre>
CH\ifa\draw\util\StorableInput.java : readStorable	
CH\ifa\draw\figures\RoundRectangleFigure.java : getArc	Calling Method - CH\ifa\draw\contrib\GraphicalCompositeFigure.java : read <pre> public void read(StorableInput dr) throws IOException { super.read(dr); setPresentationFigure((Figure)dr.readStorable()); setLayouter((Layouter)dr.readStorable()); } </pre>
CH\ifa\draw\figures\BorderTool.java : replaceAffectedFigures	
CH\ifa\draw\util\UndoManager.java : clearRedos	Calling Method - CH\ifa\draw\contrib\html\TextHolderContentProducer.java : read <pre> public void read(StorableInput dr) throws IOException { super.read(dr); ffigure = (TextHolder)dr.readStorable(); } </pre>
CH\ifa\draw\standard\BoxHandleKit.java : addHandles	
CH\ifa\draw\contrib\PolygonFigure.java : scaleRotate	Calling Method - CH\ifa\draw\standard\LocationConnector.java : read <pre> public void read(StorableInput dr) throws IOException { super.read(dr); setLocator((Locator)dr.readStorable()); } </pre>
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : translate	
CH\ifa\draw\application\DrawApplication.java : open	
CH\ifa\draw\framework\FigureAttributeConstant.java : getName	
CH\ifa\draw\util\StorableOutput.java : writeInt	
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : setClip	
CH\ifa\draw\contrib\TriangleFigure.java : rotate	
CH\ifa\draw\util\StorableInput.java : readDouble	
CH\ifa\draw\application\DrawApplication.java : exit	
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : drawImage	
CH\ifa\draw\figures\PolyLineHandle.java : setOldPoint	

Figure 17: User Clicks Cross Cutting Concern Details Link (clicked line in red).

Calls At The End Of A Method	
Called Method Summary	Called / Calling Methods Detail
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : drawPolyline	Calling Method - CH\ifa\draw\contrib\GraphicalCompositeFigure.java : read <pre> public void read(StorableInput dr) throws IOException { super.read(dr); setPresentationFigure((Figure)dr.readStorable()); setLayouter((Layouter)dr.readStorable()); } </pre>
CH\ifa\draw\util\StorableInput.java : readStorable	
CH\ifa\draw\figures\RoundRectangleFigure.java : getArc	Calling Method - CH\ifa\draw\contrib\html\TextHolderContentProducer.java : read <pre> public void read(StorableInput dr) throws IOException { super.read(dr); ffigure = (TextHolder)dr.readStorable(); } </pre>
CH\ifa\draw\figures\BorderTool.java : replaceAffectedFigures	
CH\ifa\draw\util\UndoManager.java : clearRedos	Calling Method - CH\ifa\draw\standard\LocationConnector.java : read <pre> public void read(StorableInput dr) throws IOException { super.read(dr); setLocator((Locator)dr.readStorable()); } </pre>
CH\ifa\draw\standard\BoxHandleKit.java : addHandles	
CH\ifa\draw\contrib\PolygonFigure.java : scaleRotate	
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : translate	
CH\ifa\draw\application\DrawApplication.java : open	
CH\ifa\draw\framework\FigureAttributeConstant.java : getName	
CH\ifa\draw\util\StorableOutput.java : writeInt	
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : setClip	
CH\ifa\draw\contrib\TriangleFigure.java : rotate	
CH\ifa\draw\util\StorableInput.java : readDouble	
CH\ifa\draw\application\DrawApplication.java : exit	
CH\ifa\draw\contrib\zoom\ScalingGraphics.java : drawImage	
CH\ifa\draw\figures\PolyLineHandle.java : setOldPoint	

Figure 18: The Remaining Detail for the Calling Methods is Displayed when Scrolling Down.

Summary of Results

This study showed that all cross cutting concern categories covered by this study can be successfully mined for in source code. Further, an XML cross cutting concern metadata representation was successfully implemented capturing cross cutting concerns mining results. This XML representation was used in the aspect mining and visualization process. The study showed that the three aspect mining techniques complemented each other well and was good at detecting different types of cross cutting concern categories. All three aspect mining techniques were successful in capturing sufficient information for XML metadata representations of the cross cutting concerns.

The aspect mining technique used for cloning, which is based on examining an Aspect Syntax Tree (AST) representation of the software, was found to be particularly well suited for real time aspect mining. This technique was found to be robust, have wide software system application, and to be useful when mining for a wide variety of cross cutting concerns. The cornerstone of this implementation is the *MethodRepresentation.java* class. This class was used in the method identification process and instances of this method encapsulates all methods in all classes in the JHotDraw software system. This method abstraction facilitated automated analysis of static relationships and call patterns between methods in the JHotDraw application.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

This dissertation set out to perform automated aspect mining and visualization using three different aspect mining algorithms. Mined for was a set of predetermined cross cutting concern categories. Three very different aspect mining techniques were chosen. Two static aspect mining techniques and one dynamic aspect mining technique. The study showed that aspect mining and visualization can be performed successfully in real time with a mid-sized application such as JHotDraw. A common XML representation for cross cutting concerns was developed. This is important because the XML representation separates the concern of aspect mining from that of aspect visualization. This architectural implementation is significant because it makes it possible for tool builders to decouple implementation activities and develop aspect mining and visualization software independently given a common understanding of how metadata for the cross cutting concerns is going to be represented.

Implications

This study showed that it is possible to do meaningful aspect mining in real time using an automated aspect mining tool. Further, that the results can be visualized and can be used as input for reverse software engineering activities to improve software quality by refactoring cross cutting concerns into aspects. Software developers can with this aspect mining and visualization tool reverse engineer legacy software in real time. The leading Integrated Development Environments (IDEs) for Java development IntelliJ (IntelliJ IDEA, 2018) and Eclipse (The Eclipse Foundation, 2018) already have plugins that are used for refactoring software to improve

quality. These and other IDEs support refactoring of software by extracting common code to methods and constants, global renaming of variable, methods, classes, and packages, etc. Little work has been done to reverse engineer software for increased modularity and understandability by application of aspect oriented refactoring. This study is important because it demonstrates that such reverse software engineering is possible and that this can be used to improve software quality. This tool showed further that a variety of cross cutting concern candidates can in real time be mined for, inspected, and refactored into aspects for increased software modularity and maintainability.

Recommendations

The aspect mining and visualization tool can be extended and used as a plug-in in Interactive Development Environments (IDE) such as IntelliJ and Eclipse. The tool can also be used as a standalone application deployed in a web application architecture. The purpose is in either case to aid software developers in identifying cross cutting concerns and help refactoring these into aspects through visualization. This study found that static analysis using an aspect syntax tree (AST) was most successful and easiest to apply for identification of cross cutting concerns. AST analysis is very robust, has wide application, and can detect a variety of cross cutting concern categories. The legacy software under study does not even have to compile for this analysis to be successful. All that is needed is access to the source code. Further, the tool can be used for investigation and understanding of compiled code and jar files after first decompiling these. Many decompilers are available. IntelliJ uses the IntelliJ JD plugin which does an excellent job of decompiling class files into human readable Java source code.

This study showed that aspect mining using execution traces was an excellent complimentary aspect mining technique to that of static analysis using the AST. Stricter requirements apply when using tracing. The code must compile and a set of predefined executable use cases with good code coverage must be present for aspect mining using execution traces to be effective. Organizations are increasingly adopting continuous integration and testing as part of the software development lifecycle. This makes for wider application of aspect mining using execution traces as part of the build process.

Using the aspect mining and visualization tool as part of the build process benefits the software development effort. The software development team can by applying static aspect mining (AST) and dynamic aspect mining (execution traces) discover aspect refactoring candidates during implementation of new features in the continuing integration software development process.

Summary

This study showed that it is possible to automatically mine for cross cutting concerns in legacy code and that it is possible to create a common foundation for the cross cutting concern categories and to create a representation of these in XML. This study demonstrated that cross cutting concerns can be described with XML. Two complementary cross cutting concern XML structures with corresponding DTD was developed to cover various cross cutting concern categories.

Cluster analysis of software for detection of cross cutting concerns was found to be most valuable for complex cross cutting concerns such as those based on design patterns. Cluster analysis can detect such cross cutting concerns because it is looking for the presence of a set of

methods that characterizes these types of cross cutting concerns. The cross cutting concerns of, Persistence, Decorator, Command, and Observer are all examples of such cross cutting concerns. The downside to this type of analysis from an automated aspect mining perspective is that knowledge about the cross concerns searched for must be applied to identify in which clusters these cross cutting concerns are located. This knowledge is method naming conventions and expected method call patterns.

Further, this study showed that static analysis of an aspect syntax tree representation of the software that is mined for is well suited when mining for cross cutting categories. This technique allows for full analysis of the entire code base without having to compile or run the software. This is very important because it allows for aspect mining of any Java software that can benefit from this type of analysis. Further, analysis based on an abstract syntax tree representation of software allows for mining for a variety of different types of cross cutting concern categories, not just code clones.

Dynamic aspect mining using event tracing proved to be valuable and excellent at finding cross cutting categories if the software traced satisfy the following three criteria: (1) The software must compile. (2) The software must be runnable so that execution traces can be generated when use cases are executed against the software. (3) The use cases must cover a high percentage of the software (80% or more is commonly referred to as good code coverage). For well-engineered and maintained software systems this is not a problem since these systems tend to have good code coverage from unit and integration tests that are executed against the software system in a continuous integration environment. There is a trend toward looking at software as assets in today's organizations and using continuous integration for software development is

becoming more commonplace. This makes it possible to take better advantage of aspect mining using execution traces than what has traditionally been the case.

Appendices

Appendix A

Aspect Mining and Visualization Tool Architecture

The architecture diagram for the Aspect Mining and Visualization Tool is depicted in the diagram below. This consists of two contemporary web applications and a database. The web applications are deployed in the Apache Tomcat Application Server (version 9.0.0.M20).

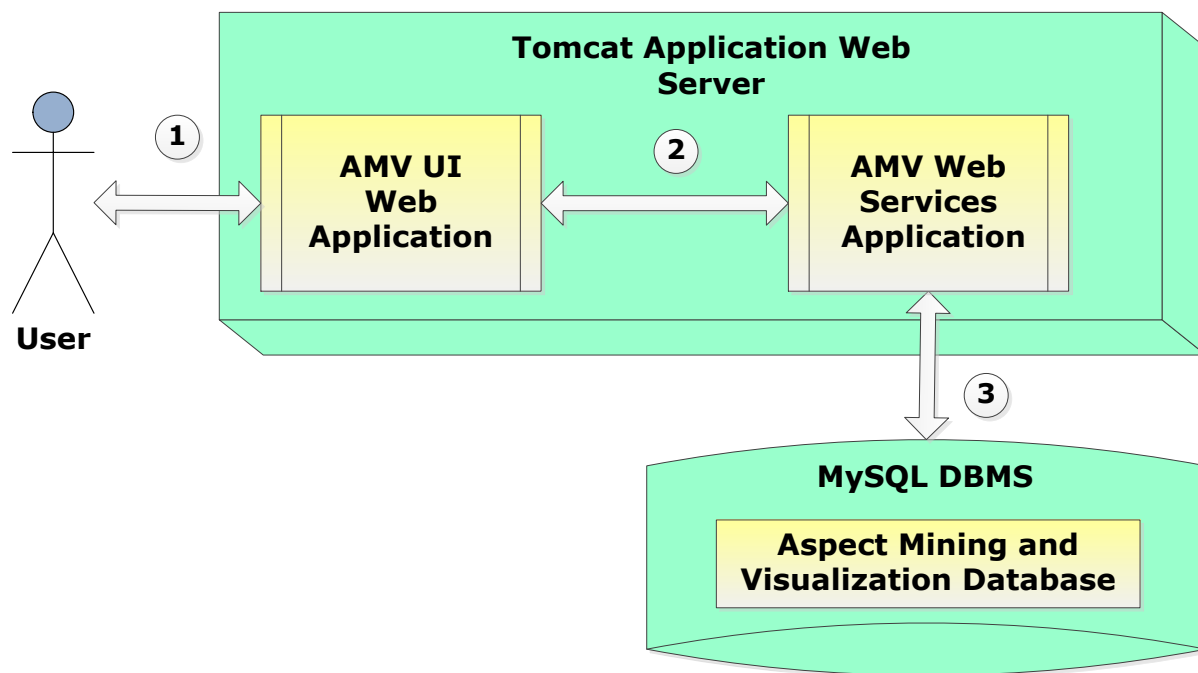


Figure 19: Aspect Mining and Visualization Application (AMV) Architecture.

- (1) The user brings up the User Interface by going to the URL in Chrome (or any other web browser) for the AMV UI Web Application.
- (2) Any requests the user submits in the AMV UI Web Application is forwarded to the AMV RESTful Web Services Web Application. This is implemented in Java using Spring WS
- (3) Data collected for individual runs are stored in the MySQL.

Appendix B

Aspect Mining and Visualization Tool Application Components

The Aspect Mining and Visualization Tool is composed of two collaborating web applications:

(1) AMV UI Web Application and (2) AMV Web Services Application.

The AMV UI Web Application (packaged in a war file) is a thin single page web application which uses HTML, AngularJS, and Bootstrap CSS. This web application communicates with AJAX calls with the AMV Web Services Application. This RESTful web application is composed of components as depicted below. This Java application uses the Maven build tool to build war and jar files:

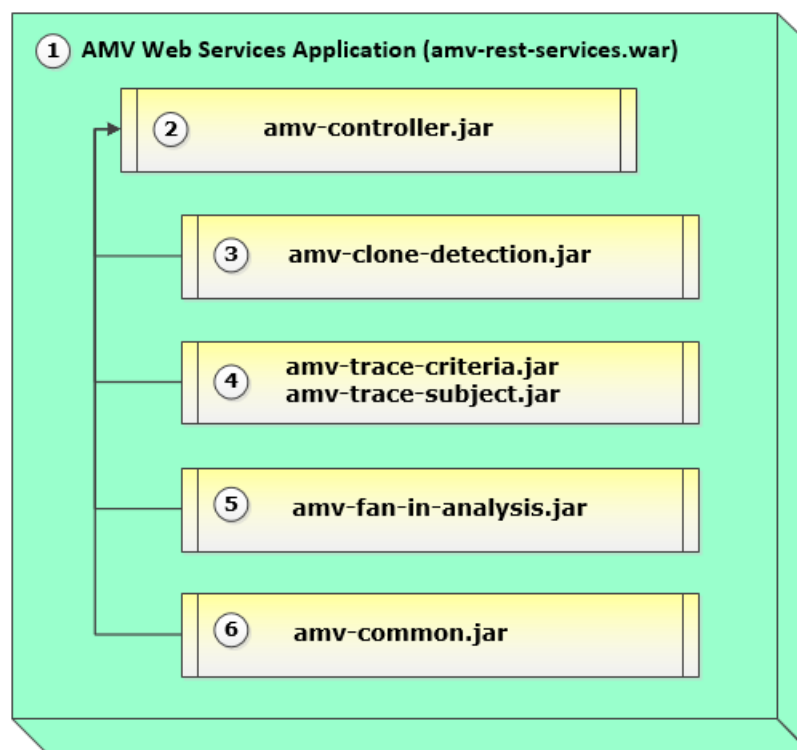


Figure 20: Aspect Mining and Visualization Tool Application Components.

(1) This war houses the RESTful APIs the AMV UI web application makes AJAX calls to.

(2) This module delegates responsibilities to (2), (3), (4), and (5) and synthesizes results.

- (3) This module performs clone detection aspect mining.
- (4) This module performs execution tracing aspect mining.
- (5) This module performs aspect mining based on fan in analysis results.
- (6) This module contains common code and data structures.

Appendix C

Aspect Oriented Programming Artifacts for Method Execution Trace Log

Figure 21 shows the aspect that was developed for tracing method calls in the JHotDraw software when executing all Junit tests that comes with the JHotDraw distribution. The pointcut expression, before advice, and after advice makes it possible to trace all method invocations during program execution and log these to a Log4J log file using indentation to show method call depth.



```

21  @Aspect
22  public class MethodExecutionTraceAspect {
23      private Logger log = Logger.getLogger(getClass().getName());
24
25      @Pointcut("execution(* *.*(..)) && !within(org.nsu.dcis.gj214.trace.criteria.MethodExecutionTraceAspect)")
26      public void traced() {}
27
28      // Before advice used for tracing method execution
29      @Before("traced()")
30      public void beforeTraced(JoinPoint joinPoint) {
31          Signature signature = joinPoint.getStaticPart().getSignature();
32          signature.toShortString();
33          log.info("Entering[" + signature + "]");
34          NDC.push(" ");
35      }
36
37      // After advice used for tracing method execution
38      @After("traced()")
39      public void afterTraced(JoinPoint joinPoint) {
40          NDC.pop();
41      }
42  }

```

The image shows a code editor with the following annotations:

- A red arrow points from the **Pointcut Expression** label to the `@Pointcut` annotation on line 25.
- A red arrow points from the **Before Advice** label to the `@Before` annotation on line 29.
- A red arrow points from the **After Advice** label to the `@After` annotation on line 38.

Figure 21: Configuration of MethodExecutionTrace Aspect.

Figure 22 shows an excerpt from the *method_execution_trace.log* log file that was produced when running all JUnit tests with the *MethodExecutionTraceAspect* included. The tracing process writes two groups of relations to the log file. Those to be included in the analysis and those to be excluded from further consideration. Relations to be excluded include those generated by the running the JUnit tests themselves. Other relations are included unless these are part of sample code that is not part of the JHotDraw core source code. Samples of these two types of relations is annotated in the excerpt from the *method_execution_trace.log*.

The screenshot shows a Notepad++ window titled "C:\log\method_trace.log - Notepad++". The window displays a log file with the following content:

```

206 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.test.figures.ScribbleToolTest.setUp()]
207 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[ScribbleTool CH.ifa.draw.test.figures.ScribbleToolTest.createInstance()]
208 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[DrawingEditor CH.ifa.draw.test.JHDTestBase.getDrawingEditor()]
209 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[AbstractTool.EventDispatcher CH.ifa.draw.standard.AbstractTool.createEventDispatcher()]
210 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[CollectionsFactory CH.ifa.draw.util.CollectionsFactory.current()]
211 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[List CH.ifa.draw.util.Collections.ydk12.CollectionsFactoryYDK12.createList()]
212 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.standard.AbstractTool.setEventDispatcher(AbstractTool.EventDispatcher)]
213 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.standard.AbstractTool.setEnabled(boolean)]
214 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[boolean CH.ifa.draw.standard.AbstractTool.isEnabled()]
215 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[boolean CH.ifa.draw.standard.AbstractTool.isEnabled()]
216 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[AbstractTool.EventDispatcher CH.ifa.draw.standard.AbstractTool.getEventDispatcher()]
217 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.standard.AbstractTool.EventDispatcher.fireToolEnabledEvent()]
218 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.standard.AbstractTool.checkUsable()]
219 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[boolean CH.ifa.draw.standard.AbstractTool.isEnabled()]
220 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[DrawingView CH.ifa.draw.standard.AbstractTool.getActiveView()]
221 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[DrawingEditor CH.ifa.draw.standard.AbstractTool.editor()]
222 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[DrawingView CH.ifa.draw.application.DrawApplication.view()]
223 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.standard.AbstractTool.setUsable(boolean)]
224 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[boolean CH.ifa.draw.standard.AbstractTool.isUsable()]
225 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[boolean CH.ifa.draw.standard.AbstractTool.isEnabled()]
226 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[DrawingEditor CH.ifa.draw.standard.AbstractTool.editor()]
227 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.application.DrawApplication.addChangeListener(ViewChangeListener)]
228 INFO [2017-11-26 10:29:08,703] [main] [MethodExecutionTraceAspect.java:33] - Entering[void CH.ifa.draw.test.figures.ScribbleToolTest.testActivate()]

```

Two annotations are present in the top right corner of the window:

- Ignored Relation: Part of JUnit Framework:** This annotation points to the first two lines of the log (lines 206 and 207).
- Included Relation: Part of JHotDraw framework:** This annotation points to the remaining lines of the log (lines 208 through 228).

The status bar at the bottom of the window shows: "Normal text file", "length: 4,249,040", "lines: 25,865", "Ln: 8", "Col: 157", "Sel: 0 | 0", "Windows (CR LF)", "UTF-8", and "INS".

Figure 22: Sample Ignore and Include Calling Relations from *method_execution_trace.log*.

The inside relations depicted in figure 14 shows the five inside relations that was identified. Four of these were called from two calling contexts whereas the fifth was called from three different calling contexts.



Figure 23: Inside Relations Identified by Automated Analysis of Generated Execution Trace log.

Figure 24 shows the *maven-shade-plugin* that was used to merge the contents from the trace-criteria and trace-subject maven modules into a single executable jar file which was used to produce the *method-trace.log* file.

```
<!-- Package up the execution tracing project, trace-subject, using the shade plugin -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <finalName>${artifactId}</finalName>
    <transformers>
      <transformer implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
        <resource>META-INF/spring.schemas</resource>
      </transformer>
      <transformer implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
        <resource>META-INF/spring.handlers</resource>
      </transformer>
    </transformers>
  </configuration>
</plugin>
```

Figure 24: Shade Plugin Used to Create Executable Jar File for Method Tracing.

The log4j.xml file depicted below contains the log4j configuration that was used when producing the method-trace.log file. Log4j was chosen because it exposes the NDC object that was used to show the method trace an indented list of method calls depicting method call depth.



```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/' >
3
4      <appender name="consoleAppender" class="org.apache.log4j.ConsoleAppender">
5          <layout class="org.apache.log4j.PatternLayout">
6              <param name="conversionPattern" value="%5p [%d] [%t] [%F:%L] - %x%m%n" />
7          </layout>
8      </appender>
9
10     <appender name="fileAppender" class="org.apache.log4j.DailyRollingFileAppender">
11         <param name="file" value="C:/log/method_trace.log" />
12         <param name="datePattern" value="'.'yyyy-MM-dd" />
13         <param name="append" value="true" />
14         <param name="threshold" value="DEBUG" />
15         <layout class="org.apache.log4j.PatternLayout">
16             <param name="ConversionPattern" value="%5p [%d] [%t] [%F:%L] - %x%m%n" />
17         </layout>
18     </appender>
19
20     <root>
21         <level value="info" />
22         <appender-ref ref="consoleAppender"/>
23         <appender-ref ref="fileAppender" />
24     </root>
25
26 </log4j:configuration>

```

Figure 25: Log4j Configuration Capturing Trace Log in C:/log/method_execution_trace.log.

Appendix D

Fan-In Results File Excerpt

Excerpt from the file generated from the Fan-In analysis with fan in threshold value of 10 has been included below.

```

C:\log\AnalysisResults.txt - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
AnalysisResults.txt
1 CH.ifa.draw.util.StorableInput.readString() : 10
2 Callers:
3 > CH.ifa.draw.figures.AbstractLineDecoration.read(QStorableInput;)
4 > CH.ifa.draw.figures.FigureAttributes.read(QStorableInput;)
5 > CH.ifa.draw.contrib.TextAreaFigure.read(QStorableInput;)
6 > CH.ifa.draw.figures.TextFigure.read(QStorableInput;)
7 > CH.ifa.draw.contrib.html.ContentProducerRegistry.read(QStorableInput;)
8 > CH.ifa.draw.figures.AttributeFigure.read(QStorableInput;)
9 > CH.ifa.draw.contrib.html.URLContentProducer.read(QStorableInput;)
10 > CH.ifa.draw.contrib.html.ResourceContentProducer.read(QStorableInput;)
11 > CH.ifa.draw.figures.ImageFigure.read(QStorableInput;)
12 > CH.ifa.draw.util.StorableInput.readStorable()
13 CH.ifa.draw.application.DrawApplication.toolDone() : 19
14 Callers:
15 > CH.ifa.draw.samples.javadraw.URLTool.mouseDown(QMouseEvent;II)
16 > CH.ifa.draw.standard.CreationTool.mouseUp(QMouseEvent;II)
17 > CH.ifa.draw.contrib.SplitConnectionTool.mouseDown(QMouseEvent;II)
18 > CH.ifa.draw.contrib.NestedCreationTool.toolDone()
19 > CH.ifa.draw.contrib.TextAreaTool.mouseUp(QMouseEvent;II)
20 > CH.ifa.draw.figures.TextTool.mouseDown(QMouseEvent;II)
21 > CH.ifa.draw.standard.ConnectionTool.mouseUp(QMouseEvent;II)
22 > CH.ifa.draw.application.DrawApplication.open(QDrawingView;)
23 > CH.ifa.draw.contrib.CompositeFigureCreationTool.toolDone()
24 > CH.ifa.draw.application.DrawApplication.promptOpen()
25 > CH.ifa.draw.standard.ActionTool.mouseUp(QMouseEvent;II)
26 > CH.ifa.draw.contrib.SplitConnectionTool.mouseUp(QMouseEvent;II)
27 > CH.ifa.draw.figures.TextTool.mouseUp(QMouseEvent;II)
28 > CH.ifa.draw.application.DrawApplication$1.run()
29 > CH.ifa.draw.figures.ScribbleTool.mouseUp(QMouseEvent;II)
30 > CH.ifa.draw.contrib.TextAreaTool.mouseDown(QMouseEvent;II)
31 > CH.ifa.draw.contrib.PolygonTool.mouseDown(QMouseEvent;II)
32 > CH.ifa.draw.application.DrawApplication.promptSaveAs()
33 > CH.ifa.draw.contrib.MDI_DrawApplication.newWindow(QDrawing;)
length: 203,212 line: Ln: 1 Col: 1 Sel: 0 | 0 Windows (CR LF) UTF-8 INS
  
```

Figure 26: Fan-In Results File Excerpt.

Appendix E

Aspect Mining Results DTD

This DTD show how cross cutting results are represented when mining for

CODE_CLONES_BEFORE_ADVICE and CALLS_AT_THE_END_OF_A_METHOD.

```

<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ASPECT-MINING-RESULT">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CROSS-CUTTING-CONCERN-CATEGORIES">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CROSS-CUTTING-CONCERN-CATEGORY" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="CROSS-CUTTING-CONCERN-CATEGORY-TYPE"/>
                    <xs:element name="CODE_CLONE_BEFORE_ADVICE" maxOccurs="unbounded" minOccurs="0">
                      <xs:element name="DUPLICATE-METHOD" maxOccurs="unbounded" minOccurs="2">
                        <xs:complexType>
                          <xs:sequence>
                            <xs:element type="xs:string" name="METHOD-NAME"/>
                            <xs:element type="xs:string" name="FILE-NAME"/>
                            <xs:element type="xs:short" name="STARTING-LINE-NUMBER"/>
                            <xs:element type="xs:short" name="ENDING-LINE-NUMBER"/>
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
                    </xs:element>
                    <xs:element type="xs:string" name="CROSS-CUTTING-CONCERN-CATEGORY-TYPE"/>
                    <xs:element name="CALLS_AT_THE_END_OF_A_METHOD" maxOccurs="unbounded" minOccurs="0">
                      <xs:element name="REFERENCED-METHOD" minOccurs="1">
                        <xs:complexType>
                          <xs:sequence>
                            <xs:element type="xs:string" name="METHOD-NAME"/>
                            <xs:element type="xs:string" name="FILE-NAME"/>
                            <xs:element type="xs:byte" name="STARTING-LINE-NUMBER"/>
                            <xs:element type="xs:byte" name="ENDING-LINE-NUMBER"/>
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
                      <xs:element name="REFERENCING-METHODS" minOccurs="2">
                        <xs:complexType>
                          <xs:sequence>
                            <xs:element name="REFERENCING-METHOD" maxOccurs="unbounded" minOccurs="1">
                              <xs:complexType>
                                <xs:sequence>
                                  <xs:element type="xs:string" name="METHOD-NAME"/>
                                  <xs:element type="xs:string" name="FILE-NAME"/>
                                  <xs:element type="xs:short" name="STARTING-LINE-NUMBER"/>
                                  <xs:element type="xs:short" name="ENDING-LINE-NUMBER"/>
                                </xs:sequence>
                              </xs:complexType>
                            </xs:element>
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>

```

Figure 27: DTD for XML Representing the Aspect Mining Results.

References

- AngularJS (2018) Web Application Development Framework, Retrieved from <https://angularjs.org/>
- Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998) Clone Detection using Abstract Syntax Trees. *International Conference on Software Maintenance*. IEEE, 368-377.
- Bootstrap (2018) Open source toolkit for developing with HTML, CSS, and JS, Retrieved from <https://getbootstrap.com/>
- Breu, S., & Krinke, J. (2004). Aspect mining using event traces. In *Ase '04: Proceedings of the 19th International Conference on Automated Software Engineering* (pp. 310–315). Washington, DC, USA: IEEE Computer Society.
- Bruntink, M., Deursen, A., Engelen, R., & Tourwe, T. (2005) On the use of clone detection for identifying crosscutting concern code. *Transactions on Software Engineering IEEE*, 804-818.
- Binkley, D., Ceccato, M., Harman, M., Ricca, F., & Tonella, P. (2005). Automated refactoring of object oriented code into aspects. *International Conference on Software Maintenance*. ICSM, 27–36.
- Ceccato, M., Tonella, P. (2009) Dynamic aspect mining. *Software, IET. IEEE*, 3(4), 321-336.
- Coelho, W., Murphy, G.C. (2006) Presenting crosscutting structure with active models. *Proceedings of the 5th International Conference on Aspect-oriented Software Development*. ACM, 158-168.
- Cojocar, G.S., & Czibula, G. (2008) On clustering based aspect mining. *Proceedings of the 4th International Conference on Intelligent Computer Communication and Processing. IEEE*, 129-136.
- Cojocar, G.S., Czibula, G., & Czibula, I.G. (2009a) A Comparative Analysis of Clustering Algorithms. *Aspect Mining. Informatica*, 75-84.
- Czibula, G., Cojocar, G. S., & Czibula, I. G. (2009b). A partitioned clustering algorithm for crosscutting concerns identification. In *Sepads '09: Proceedings of the 8th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, 111–116.
- Fabry, J., Bergel, A. (2013) Design decisions in AspectMaps. *Software Visualization (VISSOFT)*. IEEE, 1-4.

- Fabrya, J., Kellensb, A., Denierc, & S., Ducasse, S (2012) AspectMaps: Extending Moose to visualize AOP software. *Science of Computer Programming*. 79, 6-22.
- Fabry, J., Kellens, A., & Ducasse, S. (2011) AspectMaps: A Scalable Visualization of Join Point Shadows. *19th International Conference on Program Comprehension (ICPC)*. IEEE, 121 - 130.
- Fowler, Martin (1999) *Refactoring: Improving the Design of Existing Code*. Boston, Massachusetts, Addison-Wesley.
- Ferrante, J., Ottenstein, K., J., & Warren, J., D. (1987) The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems*. ACM, 319-349.
- Fraley, C. & Raftery, A. E. (2006) MCLUST Version 3 for R: Normal Mixture Modeling and Model-Based Clustering. *Technical Report 504, University of Washington*, 1-56.
- Griswold, W.G., Yuan, J.J., & Kato, Y. (2001) Exploiting the map metaphor in a tool for software evolution. *Proceedings of the 23rd International Conference on Software Engineering*. IEEE, 265-274.
- Hannemann, J., & Kiczales, G. (2001). Overcoming the Prevalent Decomposition in Legacy Code. *Proceedings of the ICSE Workshop on Advanced Separation of Concerns*. Retrieved from <http://www.research.ibm.com/hyperspace/workshops/icse2001/Papers/hannemann.pdf>
- Hannemann, J., Kiczales, G. (2002) Design pattern implementation in Java and AspectJ. *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 161-173.
- Hacoupan, Y. (2013). *Mining Aspects through ClusterAnalysis Using Support Vector Machines and Generic Algorithms*. (Doctoral dissertation). Retrieved from ProQuest Dissertations and Theses. (Accession Order No. 3567556).
- IntelliJ IDE (2018) The Java IDE for Professional Developers Retrieved from <https://www.jetbrains.com/idea/>
- Jain, A.K, Murty, M.N., Flynn, P.J. (1999) Data clustering: A Review. *ACM Computer Surveys*. ACM, 264-323.
- Janzen, D., & Volder, K. (2003) Navigating and Querying Code Without Getting Lost. *2nd International Conference on Aspect-oriented Software Development*. ACM
- Komondoor, R., Horwitz, S. (2001) Using Slicing to identify Duplication in Source Code. *Static Analysis (2001)*.
- Kontogiannis, K. , R. Demori, R., Merlo, E., Galler, E., and Bernstein, M. (2003). Evaluation

- Experiments On the Detection of Programming patterns Using Software Metrics. *In Proceedings of the 2nd Working Conference on Reverse Engineering*, pp. 96–103.
- Krinke, J. (2006). Mining Control Flow Graphs for Crosscutting Concerns. *Proceedings of the 2nd Working Conference on Reverse Engineering*.
- Krinke, J. (2001). Identifying Similar Code with Program Dependency Graphs. *Proceedings of the Eight Working Conference on Reverse Engineering*.
- Laddad, R. (2003) AspectJ in Action: Practical Aspect-Oriented Programming. Chap.8, Chap.10 Greenwich, CT, USA: Manning Publications Co.
- Maisikeli, S.G., Mitropoulos, F.J. (2010) Aspect mining using Self-Organizing Maps with method level dynamic software metrics as input vectors. *Proceedings of 2nd International Conference on Software Technology and Engineering (ICSTE)*. IEEE, 212-217.
- Martin, M., Deursen, A., & Moonen, L. (2004). Identifying Aspects using fan-in analysis. *Proceedings of the 11th Conference on Reverse Engineering*. 132-141.
- Martin, M., Moonen, L., & Deursen, A. (2006). FINT: Tool Support for Aspect Mining. *Proceedings of the 13th Working Conference on Reverse Engineering*, 299-300.
- Mayrand, J.; Leblanc, C.; Merlo, E.M. (1996) Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. *International Conference on Function Clones in a Software System Using Metrics*. IEEE, 244-253.
- Mens, K., Kellens, A., & Krinke, J. (2008) Pitfalls in Aspect Mining. *Proceedings of the 15th Working Conference on Reverse Engineering*. IEEE, 113-122.
- Moldovan, G. S., & Serban, G. (2006) Aspect mining using a vector-space model based clustering approach. *Proceedings of Linking Aspect Technology and Evolution (late) workshop*. 36–40, Bonn, Germany.
- Monteiro, M. P., Fernandes, J. M. (2004) Object-to-aspect refactorings for feature extraction. *International Conference on Aspect-Oriented Software Development*.
- Pfeiffer, J.H., Gurd, J.R. (2006) Visualization-based tool support for the development of aspect-oriented programs. *Proceedings of the 5th International Conference on Aspect-oriented Software Development*. ACM, 146-157
- Rand McFadden, R.R., & Mitropoulos, F.J. (2012) Aspect mining using model-based clustering. *2012 Proceedings of Southeastcon*. IEEE, 1-8.
- Rand McFadden, R.R., & Mitropoulos, F.J. (2013a) Survey of Aspect Mining Case Study Software and Benchmarks. *2013 Proceedings of Southeastcon*. IEEE, 1-5.

- Rand McFadden, R.R., & Mitropoulos, F.J. (2013b) Survey and Analysis of Quality Measures Used in Aspect Mining. *2013 Proceedings of Southeastcon*. IEEE, 1-8.
- Rand McFadden, R.R., & Mitropoulos, F.J. (2013) Survey of Aspect Mining Case Study Software and Benchmarks. *2013 Proceedings of Southeastcon*. IEEE, 1-5.
- Robillard, M., Murphy, G.C. (2002) Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. *Proceedings of the 24th International Conference on Software Engineering*. ACM, 406-416.
- Serban, G., & Moldovan, G.S. (2006a) A New k-means Based Clustering Algorithm in Aspect Mining. *Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 69-74.
- Serban, G., & Moldovan, G. (2006b) A New Genetic Clustering Based Approach in Aspect Mining. *Proceedings of the 8th WSEAS International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering*. 135-140.
- Shepherd, D., Gibson, E., & Pollock, L. (2004). Design and evaluation of automated aspect mining tool. *Software Engineering Research and Practice*, 601–607.
- Shepherd, D., Palm, J., & Pollock, L. (2005). Timna: A Framework for Automatically Combining Aspect Mining Analysis. *Proceedings of the 20th international Conference on Automated Software Engineering* ACM, 184-193.
- Shepherd, D., & Pollock, L. (2005) Interfaces, Aspects, and Views. *Workshop on Linking Aspect Technology and Evolution (LATE 2005), Co-located with International Conference on Aspect Oriented Software Development*.
- Shepherd, D., Pollock, L., & Tourwe, T. (2005) Using language clues to discover crosscutting concerns. *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software*. ACM, 1-6
- The Eclipse Foundation. (2018). *Eclipse*. Retrieved from <http://www.eclipse.org/aspectj/>
- Tonella, P., & Ceccato, M. (2004). Aspect mining through the formal concept analysis of execution traces. *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, 112–121.
- Tribbey, W., Mitropoulos, F., (2012) Construction and analysis of vector space models for use in aspect mining. *Proceedings of the 50th Annual Southeast Regional Conference*. ACM, 220-225.
- Van Bruggen, Danny (2017) Javaparser [Software] Available from

<https://github.com/matozoid/javaparser>

Zhang, D., Guo, Y., Chen, X. (2008) Automated Aspect Recommendation through Clustering-Based Fan-in Analysis. *23rd International Conference on Automated Software Engineering*. IEEE, 278 – 287

Zhu, J., Yin, Q., Zhu, R., Guo, C., Wang, H., Wu, Q (2008) A Plugin-Based Software Production Line Integrated Framework. *International Conference on Computer Science and Software Engineering*. IEEE, 562-565.