2015

Locating Potential Aspect Interference Using Clustering Analysis

Brian Todd Bennett
Nova Southeastern University, bennettbtodd@gmail.com

Locating Potential Aspect Interference Using Clustering Analysis

by

Brian Todd Bennett

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

Graduate School of Computer and Information Sciences
Nova Southeastern University

2015

We hereby certify that this dissertation, submitted by Brian Bennett, 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. Date
Chairperson of Dissertation Committee

_____ _____

Sumitra Mukherjee, Ph.D. Date
Dissertation Committee Member

_____ _____

Renata Rand McFadden, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Eric S. Ackerman, Ph.D. Date
Dean, Graduate School of Computer and Information Sciences

Graduate School of Computer and Information Sciences
Nova Southeastern University

2015

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Locating Potential Aspect Interference Using Clustering Analysis

by
Brian Todd Bennett
May 2015

Software design continues to evolve from the structured programming paradigm of the 1970s and 1980s and the *object-oriented programming* (OOP) paradigm of the 1980s and 1990s. The functional decomposition design methodology used in these paradigms reduced the prominence of non-functional requirements, which resulted in scattered and tangled code to address non-functional elements. *Aspect-oriented programming* (AOP) allowed the removal of *crosscutting concerns* scattered throughout class code into single modules known as *aspects*. Aspectization resulted in increased modularity in class code, but introduced new types of problems that did not exist in OOP. One such problem was *aspect interference*, in which aspects meddled with the data flow or control flow of a program. Research has developed various solutions for detecting and addressing aspect interference using formal design and specification methods, and by programming techniques that specify aspect precedence. Such explicit specifications required practitioners to have a complete understanding of possible aspect interference in an AOP system under development. However, as system size increased, understanding of possible aspect interference could decrease. Therefore, practitioners needed a way to increase their understanding of possible aspect interference within a program. This study used *clustering analysis* to locate potential aspect interference within an aspect-oriented program under development, using *k-means* partitional clustering. Vector space models, using two newly defined metrics, *interference potential (IP)* and *interference causality potential (ICP)*, and an existing metric, *coupling on advice execution (CAE)*, provided input to the clustering algorithms. Resulting clusters were analyzed via an internal strategy using the *R-Squared*, *Dunn*, *Davies-Bouldin*, and *SD* indexes. The process was evaluated on both a smaller scale AOP system (AspectTetris), and a larger scale AOP system (AJHotDraw). By seeding potential interference problems into these programs and comparing results using visualizations, this study found that clustering analysis provided a viable way for detecting interference problems in aspect-oriented software. The ICP model was best at detecting interference problems, while the IP model produced results that were more sporadic. The CAE clustering models were not effective in pinpointing potential aspect interference problems. This was the first known study to use clustering analysis techniques specifically for locating aspect interference.

Acknowledgements

Dedication

This work is dedicated in loving memory of my father,
Leonard Eugene Bennett,
who showed me how to write my first program at age five.
I love you, I miss you, and I wish you could see this.

The dissertation journey once seemed a long, winding road with its end remaining over a distant hill. With each turn came new learning opportunities in research and statistical techniques that have led to the end of this road. Looking back, I reflect on the many people who helped this journey reach its conclusion.

I thank my advisor, Dr. Frank Mitropoulos, for introducing me to this area of research, and for sending me in the right direction early in the process. I appreciate the prompt and constructive comments and feedback received from my committee members, Drs. Sumitra Mukherjee and Renata Rand McFadden. Your support has been greatly appreciated.

I thank my professors and mentors at East Tennessee State University who instilled a strong foundation for software engineering that has carried me through my professional career to this point, and is sure to carry me throughout the rest of my life.

I thank my family members who were there for me throughout this process. I especially thank my mother, Coleen Bennett, for her unwavering support, unconditional love, and unmatched faith. Through her continual encouragement and her example, navigating this and all of life's journeys is much easier. I love you.

Finally, above all, I thank God through Jesus Christ, from whom all blessings flow. Grace to stay the course, ability to complete the task, guidance, encouragement, family, life—all things come from Him. To God be the glory! With God, all things are possible.

Table of Contents

List of Tables

Tables

List of Figures

Figures

Chapter 1

Introduction

Background

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) has become a mainstream paradigm in software engineering that originated in the 1970s from structured programming. Dahl and Hoare (1972) noted that structured concepts such as types, variables, and arrays involved both data structures and procedures that operated upon them. From these observations, Dahl and Hoare (1972) introduced the concept of a *class*—a structure that encapsulated both variables and procedures—using the SIMULA 67 programming language. Creating a structure that encapsulated both data and procedures gave a new way to model real-world functionality within software systems. Because of the ability to modularize programs in this manner, object-oriented languages like Smalltalk and C++ developed throughout the 1970s and 1980s (Capretz, 2003).

The modular nature of OOP also allowed program designers to view *objects*—individual instances of a class—from an external viewpoint (Rentsch, 1982). The principle of *encapsulation* noted that objects communicated through strictly defined external interfaces that hid implementation details (Rentsch, 1982; Snyder, 1986). These external interfaces protected the internal member data against modification, and allowed rewriting a class with minimal impact to a program, as long as the external interface

remained constant. However, OOP also supported the idea of *inheritance*, in which child objects could use both member data and operations defined within a parent object. Because inherited objects needed no external interface to their parent, inheritance required programmers to consider objects from an internal viewpoint to define what member data and methods child objects could utilize and manipulate. (Snyder, 1986)

The principles of encapsulation and inheritance had a major impact on software development methodology. Early OOP design approaches simply retrofit familiar structural methodologies by adding OOP design techniques to them (Capretz, 2003). Capretz (2003) noted that the ideal OOP design methodology should focus on OOP across all phases of development. Booch (1986) defined a new object-oriented design methodology that addressed limitations in classical approaches and focused on objects from both an internal and external viewpoint. By using this method, designers concentrated on the functionality of the system from the class or object perspective (Booch, 1986).

Kiczales et al. (1997) noted that most methodologies like Booch (1986) used *functional decomposition*—designing a system by breaking it into chunks of related functionality. Functional decomposition, while comfortable from an encapsulation standpoint, failed to address non-functional features of software. Because existing language constructs failed to address non-functional features, programmers codified these features inside other existing modules. Kiczales et al. (1997) referred to the mixing of one software feature within an existing module *tangling*, causing potential difficulties with maintenance. Tangling non-functional and functional features was an example of *crosscutting*—a situation where two software properties coexisted, yet each came from a

different functional composition (Kiczales et al., 1997). The authors described *crosscutting concerns* as software features tangled within one or more existing modules of a software system (Kiczales et al., 1997). Kiczales et al. (1997) observed the need to detangle crosscutting concerns from class code, and developed a new paradigm called *aspect-oriented programming* (AOP) that allowed programmers to view crosscutting concerns as modules. (Kiczales et al., 1997)

Aspect-Oriented Programming (AOP)

The goal of AOP was to increase the separation of crosscutting concerns within a program. Kiczales et al. (1997) described two distinct entities in a software system: components and aspects. A *component* was any software element that could exist as a well-defined encapsulation, such as procedures and classes. An *aspect* was a programming element that could not exist as a well-defined encapsulation in programming languages of the time. To separate aspects tangled within components, Kiczales et al. (1997) proposed a new programming concept that would place an aspect in a single location and then incorporate it into component code where needed. The process of incorporating aspect code into component code was termed *aspect weaving* (Kiczales et al., 1997).

The *aspect weaver*—a system designed to inject aspect code into component code at either runtime or compile time—needed to know the specific locations for code injection (Kiczales et al., 1997). As a result, aspect languages like AspectJ ("The AspectJ™ Programming Guide," 2003) developed mechanisms called *point cuts*, which described a set of *join points*—locations within the code that would contain aspect code after weaving (Stoerzer & Graf, 2005). *Advice* also supported point cuts by specifying

what aspect code to execute *before*, *after*, or *around* the join point ("The AspectJ™ Programming Guide," 2003; Stoerzer & Graf, 2005). Because of these mechanisms, AOP increased modularity in component code because it removed crosscutting concerns, and allowed component code to remain oblivious to the existence of aspect code within the program (Hannemann & Kiczales, 2002).

In an early assessment of AOP, Walker, Baniassad, and Murphy (1999) saw that in some cases, developers could have difficulty understanding an AOP program. These authors concluded that aspect-oriented languages needed a way to show the scope of an aspect's effect on component code, and also to encapsulate crosscutting concerns appropriately (Walker et al., 1999). However, the process of aspect weaving could cause semantic problems in the program that made the scope of an aspect's effect difficult to define and detect (Tian, Cooper, & Zhang, 2010). Aspect weaving problems could result from an aspect that interfered in a program's context, control flow, structure, or construction (Tian, Cooper, & Zhang, 2010).

Aspect Interference

Aspect interference (or interaction) was coined to describe an aspect weaving problem in which an aspect caused unexpected changes to the flow of a class or method (Douence, Fradet, & Südholt, 2002). Tian, Cooper, Zhang, and Yu (2009) noted that aspects could cause a number of syntactic and semantic problems in the woven code. Syntax problems involved issues in the programming, such as incorrect naming, and issues with point cut definitions (Tian et al., 2009). Semantic conflicts involved inconsistencies introduced by weaving, aspects that executed out of order, aspects with circular dependencies, interference with a system's functionality, behavioral and OO

composition problems, and problems when superimposing structure (Tian et al., 2009). An aspect was also able to introduce new code locations upon which other aspects would operate, break the system structure, or interact with other aspects attempting to access a common join point (Tian, Cooper, Zhang, & Liu, 2010).

Such semantic problems could occur in a software system even when the code is syntactically correct (Tian et al., 2009). In addition, even when class code and aspect code were verified during modular testing, integration could introduce unexpected behaviors produced by aspects (Delamare & Kraft, 2012). Therefore, aspect interference has remained a silent threat to programmers developing aspect-oriented software systems by causing issues that are both difficult to find and difficult to correct.

Problem Statement

Programmers faced the problem of having no way to conceptualize aspect interference fully while developing an aspect-oriented software system. Research has worked to both prevent and detect aspect interference at design time and execution time. Existing design-time detection methods called for programmers to specify the requirements or design using formal definitions that enforce non-interference (Chen, Ye, & Ding, 2010; Disenfeld & Katz, 2012, 2013; Hannousse, Douence, & Ardourel, 2011). Existing execution-time detection methods required programmers to define how aspects and advices should interact explicitly within the code via precedence definitions (Lauret, Fabre, & Waeselynck, 2011; Lauret, Waeselynck, & Fabre, 2012; Marot & Wuyts, 2009, 2010) or the implementation of monads and membranes (Figueroa, 2013; Figueroa, Tabareau, & Tanter, 2013).

Shaw (1989) noted that larger systems required additional levels of abstraction to

simplify understanding, because large systems caused the programmer's understanding of the system to decrease. Thus, when developing a large aspect-oriented software system, programmers could easily overlook formal design or precedence definitions necessary to prevent aspect interference. Only one known study investigated aspect interference using static analysis by using program slicing techniques on woven Java bytecode (D'Ursi, Cavallaro, & Monga, 2007). These authors showed that static analysis had the potential to highlight interference problems, but was unsuccessful using simplistic program slicing techniques (D'Ursi et al., 2007). Thus, the state-of-the-art provided no solution allowing programmers to locate potential aspect interference inside an aspect-oriented program without first requiring additional constructs in the design or code. This study provided a method that analyzed programs statically to allow programmers the opportunity to correct areas of potential interference in a system before go-live and during maintenance.

Dissertation Goal

The goal of this research was to give programmers a way to locate potential aspect interference during the development process, without requiring special programming or design techniques. This research investigated a clustering analysis method for examining static AOP code to identify aspect interference candidates. *Clustering analysis* emerged as a data mining technique that divided a set of predefined objects into groups exhibiting similar characteristics (Şerban & Moldovan, 2006). Clustering techniques have modeled software in a variety of ways, including simple metric-based models and vector space models. Vector space models showed promise when analyzing software because they provided more detail than clustering with simple metrics. This study introduced vector state models and fed them into the clustering

algorithm. One vector space model was based on a familiar metric—*coupling on advice execution* (CAE) (Ceccato & Tonella, 2004; Piveta et al., 2012), while the other described two newly defined metrics designed to account for method-method, advice-method, and advice-advice interactions. *K*-means clustering algorithm results revealed which vector state model produced the best detection of interference problems. Analysis involved two known aspect-oriented programs—AspectTetris and AJHotDraw—to determine both feasibility and scalability. Seeding potential aspect interference into modified versions of each application showed the technique's ability to detect a potential threat. The research also used a visualization technique to display the results of the clustering analyses graphically for assessing the potential interference problems in the software.

Research Questions

The main hypothesis of this research was that clustering analysis would provide a way for programmers to locate potential aspect interference within an existing aspect oriented software system. To support this hypothesis, the study investigated the following research questions.

1. Existing OO metrics used in clustering analysis, such as fan-in value (FIV) and fan-out value (FOV), were inadequate for describing aspect interference because they denoted only method-method interactions. FIV counted the number of methods that called a method, while FOV counted the number of methods that a method called. Similarly, existing advice metrics such as coupling on advice execution (CAE) only accounted for module-aspect interaction (Piveta et al., 2012). Could this study define new metrics that adequately described the potential for interference, and account for method-method, advice-method, and advice-

advice interactions?

2. Was clustering analysis able to detect potential aspect interference within an aspect-oriented program? Could clustering locate both advices with the potential to interfere, and the advices or methods that may be victims of interference?
3. Was clustering analysis designed to pinpoint aspect interference problems able to scale from a smaller program to a larger program with similar results?

Relevance and Significance

This study was relevant because it provided programmers who use AOP an effective way to locate potential aspect interference problems in code. AOP, while not a new paradigm, has continued to evolve as a field of research and as an accepted programming practice. Programming language support for and education in AOP will increase as research continues. In addition, aspect interference was previously seen as difficult to describe, and even harder to prevent (Tian, Cooper, & Zhang, 2010; Tian, Cooper, Zhang, et al., 2010). Processes that allowed programmers to better understand potential problems with a system under development have been an important area of research (Cassell, Anslow, Groves, & Andreae, 2011; De Borger, Lagaisse, & Joosen, 2009; Dietrich, Yakovlev, McCartin, Jenson, & Duchrow, 2008; Fabry, Kellens, & Ducasse, 2011; Lanza & Marinescu, 2006; Wettel & Lanza, 2008; Yin, 2013; Yin, Bockisch, & Aksit, 2012). Therefore, this new method for finding potential aspect interference in an AOP system was highly relevant to AOP research.

This research was significant to the body of knowledge in four ways. First, this study required no additions to a program's code to detect potential aspect interference. Previous studies have detected aspect interference at design time and execution time, but

have required either formal definitions of aspects in the design (Chen et al., 2010; Disenfeld & Katz, 2012, 2013; Hannousse et al., 2011), aspect and advice precedence definitions in the code (Lauret et al., 2011; Lauret et al., 2012; Marot & Wuyts, 2009, 2010), or definitions of monads and membranes (Figueroa, 2013; Figueroa et al., 2013). Only one study attempted to locate interference in woven Java bytecode by using program slicing, but was unsuccessful (D'Ursi et al., 2007).

Second, this was the first known study to apply clustering analysis specifically to aspect interference. Previous studies using clustering with AOP have performed aspect mining to locate crosscutting concerns in object-oriented code (G. Czibula, Cojocar, & Czibula, 2009; Moldovan & Şerban, 2006; Rand McFadden & Mitropoulos, 2012; Şerban & Moldovan, 2006; Shepherd & Pollock, 2005; Tribbey & Mitropoulos, 2012). No previous study used clustering analysis to examine aspect-oriented code for aspect interference.

Third, this study enhanced the state-of-the-art by introducing and analyzing new AOP metrics that gauged the potential for aspect interference. To understand both advice and weaving interference fully, the metrics needed to describe method-method, advice-method, and advice-advice interactions. Adding method-method interaction to advice-method and method-method interactions helped determine the magnitude of interference since methods with higher coupling could affect many parts of the system. Advice-method interaction determined potential weaving interference, while advice-advice interaction determined potential advice interference. Existing metrics accounted for only one of these three types of interaction at a given time. For example, OO metrics FIV and FOV only counted method-method interactions, and CAE only counted class-advice

interactions. This study proposed two new metrics to describe method-method, advice-method, and advice-advice interactions. The first, *interference potential of an object* (IP), based on FOV, described an object's potential to interfere with other objects by counting the number of objects it invoked. The second, *interference causality potential for an object* (ICP), based on FIV, showed the potential interference caused when other objects to invoke the object. Note that an object would be either a method of a class or an advice defined in an aspect.

Finally, to enhance understanding and aid in assessment, the results of the clustering analysis appeared visually. While visualization was not the main goal of this study, it helped increase understanding of design problems in software throughout the literature. Previous studies in visualization have involved pinpointing OO design problems (Lanza & Marinescu, 2006; Wettel & Lanza, 2008), refactoring OO design problems (Cassell et al., 2011; Dietrich et al., 2008), and exploring AOP visualizations in the context of a debugger (DeBorger, Lagaisse, & Joosen, 2009; Yin et al., 2012). Fabry, Kellens, and Ducasse (2011) created an AOP visualization designed to help programmers visualize join point interactions, but not interference. This study extended this idea to clustering results.

Barriers and Issues

One barrier to completing this research was locating aspect-oriented programs that tested both the feasibility and scalability of the proposed approach. Feasibility testing required a smaller program into which potential interference could be seeded, while scalability testing required a larger system. AOP testing research has used a variety of programs for verifying their studies. Lauret, Waeselynck, and Fabre (2012) used a small

program with three aspects that performed logging, encryption, and authentication. Delamare and Kraft (2012) used a model-view-controller bank application implemented in AspectJ. Other studies used small motivating examples rather than real-world systems to illustrate AOP testing methodology (Jianjun, 2003; Pan & Song, 2012). The work of Apel (2010), who studied 11 AspectJ programs for usage patterns, provided insight for overcoming this barrier and selecting appropriate AOP programs—AspectTetris and AJHotDraw.

A second barrier to this research was determining whether the two proposed metrics successfully described both objects that cause interference and objects impacted by other objects. Interference potential of an object (IP) was based on FOV, while interference causality potential for an object (ICP) was based on FIV. The newly defined IP metric described interference tangling by showing how many objects the object invoked, while the ICP metric described interference scattering by showing how many times other objects invoked the object. Because these were new metrics, their validity was essential for completing this study.

A third barrier was determining join point locations from the code. Correctly determining the join point locations was crucial for correctly producing valid IP and ICP metric values. Point cuts in AspectJ could map to zero or more join point locations in the code. Determining join points in source code was difficult because point cuts used regular expressions to describe join points. Because of this, examining Java bytecode produced by weaving provided a standard way of seeing advice-advice, advice-method, and method-method interaction. However, distinguishing between advices and methods in bytecode was not straightforward. Using constructs within the Ruby programming

language to separate program features helped overcome this barrier.

Limitations

This study suffered from two limitations. First, the body of literature regarding aspect interference focused mainly on including additional features in a design or code base that prevented interference from occurring. Only one study showed support for detecting aspect interference by using static code analysis, though the authors concluded that the methodology used—simple program slicing—was not adequate for locating interference (D'Ursi et al., 2007). Therefore, no other static analysis studies existed that would allow for experimental comparison.

The study was also limited by existing aspect-oriented metrics. None of the metrics in the literature accounted for fine-grained interactions at the method-method, method-advice, and advice-advice levels. Such metrics were required for the successful implementation of this study. Thus, the two new metrics overcame this limitation by providing a sum of counts representing all of these interaction levels. However, because they were new, these metrics limited this work to comparisons with existing AOP coupling metrics that had less granularity.

Definition of Terms

Advice	A member of an aspect that defines the code to inject before, after, or around a join point.
Advice Interference	Multiple advices that interact around a common join point.
Aspect	A programming element encapsulates code originally scattered throughout a program into a single location.
Aspect Interference	Conflicts between aspects.

Aspect Weaving	The process of inserting aspect code at join points at compile-time.
AspectJ	An extension of the Java language that allows aspect-oriented programming.
Aspect-Oriented Programming	A programming paradigm that views crosscutting concerns as code modules.
Class	A structure encapsulating both variables and procedures.
Clustering Analysis	Determining relationships within a set of data in a way that groups similar objects together.
Code Scattering	A crosscutting concern that exists in various locations across a system.
Code Tangling	The mixing of a crosscutting concerns with other concerns in a system.
Crosscutting Concern	A software feature that must coexist within one or more modules of a software system.
Fan-In-Value	An object-oriented metric for a method that counts the number of methods invoking that method.
Fan-Out-Value	An object-oriented metric for a method that counts the number of methods invoked by that method.
Functional Decomposition	Designing a system by breaking it into pieces of related functionality.
Inheritance	A tenet of object-oriented programming in which child objects may use both member data and operations defined within a parent object.
Interference Causality Potential	An aspect-oriented metric defined by this study that counts the number of advices or methods that invoke the given advice or method.

Interference Potential	An aspect-oriented metric defined by this study that counts the number of advices and methods invoked by the given advice or method.
Interference Scattering	The property of an aspect-oriented program that includes aspect interference spreading across a system.
Interference Tangling	The property of an aspect-oriented program that includes multiple interferences that exist on a given advice or method.
Join Point	A location within source code that will contain aspect code after weaving.
Join Point Shadow	A location within source code that will become a join point upon program execution.
k-means Partitional Clustering	A clustering algorithm that partitions a data set into k clusters based on specific characteristics of the data being analyzed.
Object-Oriented Programming	A programming paradigm that uses classes and objects within the design of a system.
Objects	(OOP) An unique instance of a class. (Clustering) A component that is fed into a clustering model.
Point Cut	A part of an aspect that describes the location of join points within source code.
Precedence	A programming technique within AspectJ that allows the programmer to specify the order of aspect code execution.
Vector Space Model	A method of input to a clustering model that includes more details that may be hidden when using simple metrics.
Weaving Interference	A type of aspect interference that originates from the weaving of aspect code into class code.

Summary

Since the advent of functional and object-oriented programming techniques, design has focused primarily on system functionality instead of non-functional elements. As a result, certain concerns needed to be distributed across and into modules of programming systems. Aspect-oriented programming allowed practitioners the ability to centralize these crosscutting concerns into aspects that advised a program of where and under what circumstances to weave the concern into the code. However, because of the control required by aspects, aspect code could interfere with the modules they interacted with, causing unexpected changes to a program's flow.

Aspect interference has continued to be an area of interest to researchers. Many researchers have attempted to prevent aspect interference by formally describing a program's design, or by adding detailed precedence rules to a program. While these techniques could help, they required programmers to have a full understanding of potential aspect interference that existed in a system under development. No research has successfully used static programming analysis to locate aspect interference.

Clustering analysis appeared in the literature as a static analysis technique for finding potential ways to refactor object-oriented software. Others have used clustering analysis to locate potential aspects within existing object-oriented code. Because of this, this dissertation extended clustering techniques to aspect-oriented programming to find potential aspect interference within existing programs.

Chapter 2

Review of the Literature

Aspect Interference

The literature showed several distinct research trends regarding aspect interference: defining aspect interference, detecting interference at design time, detecting interference at implementation time, and detecting interference at execution time. Detection first required an unambiguous definition of aspect interference. Having a concrete definition was considered extremely important when attempting to locate instances of aspect interference during software design, development, and runtime.

Defining Aspect Interference

Douence, Fradet, and Südholt (2002) provided one of the earliest studies on aspect interference, which they termed aspect interaction. The authors defined aspect interaction as conflicts between non-orthogonal aspects (Douence et al., 2002, p. 173). Although programmers were responsible for finding and correcting aspect interactions, they previously had no support for accomplishing these important tasks. Douence et al. (2002) therefore suggested a three-phase implementation model for developing AOP systems: independent component development, conflict analysis using automated testing, and conflict resolution to address interactions found in the previous phase. To accommodate these phases, the authors provided a language-independent model that included a formal language and definition set. This model included the definitions of two

new concepts: strong independence (aspects remain independent when woven into any program), and independence with respect to a program (aspects depend on join points within a specific program). The authors used these formal definitions to provide analysis, concluding that the resulting commands for conflict resolution would be useful in practice. However, the authors only began an implementation of the framework in AspectJ rather than fully developing a solution. (Douence et al., 2002)

Tian, Cooper, Zhang, and Liu (2010) readdressed the understanding of aspect interference, because they believed that the definition remained weak. They contended that existing definitions missed an advice's cumulative effect on a program and did not account for advices interfering at common join points. Additionally, the authors stated that the definition of advice interference overlapped with weaving interference. The authors presented a semi-formal definition of aspect interference using three specific terms—introduction interference, weaving interference, and advice interference. Introduction interference occurred when one aspect added or deleted code locations addressed by other aspects. Weaving interference occurred between aspects or between an aspect and the base program, resulting in violations of the system structure. Advice interference referred to advices that interacted around a shared location in class code. The authors concluded that these definitions improved the understanding, reasoning, and recognition of aspect interference. In addition, the authors more clearly separated the definitions of advice and weaving interference. (Tian, Cooper, Zhang, et al., 2010)

Tian, Cooper, and Zhang (2010) took a broader approach by defining a taxonomy of seven types of aspect weaving problems (AWPs). These included fragile contextual assumptions, fragile control flow assumptions, fragile structural assumptions, contextual

interference, control flow interference, structural interference, and introduction interference. Of the seven types, the study classified only contextual interference, control flow interference, and structural interference as forms of aspect interference. Based on these definitions, the authors developed a complex static framework that could describe a system and detect the AWPs identified. They concluded that this taxonomy was a basis for extension in future research, and would prove useful for practitioners if used in tools such as FDAF (an AOP support and design analysis tool). (Tian, Cooper, & Zhang, 2010)

Bernardi and Di Lucca (2010) created a metric model for aspect coupling which defined three different types of aspect interactions. First, aspect coupling occurred when an aspect alters the static structure of a module (CSS). This would include adding constraints, adding members, forcing the implementation of an interface, or by altering inheritance relationships. Second, aspect coupling occurred when an aspect alters the control flow of a program (CCF), accomplished by adding, replacing, or conditionally replacing code. Finally, aspect coupling occurred when an aspect alters the state of another object (COS), whether changing state values or not, or simply observing them. The authors applied their model to AJHotDraw as a feasibility case study, showing it identified coupling and interaction without evaluating the models' effectiveness. The case study revealed a much higher CSS than either CCF or COS combined, but these results may not be typical for all systems. (Bernardi & Di Lucca, 2010)

The current study explored advice interference and weaving interference as defined by Tian, Cooper, Zhang, et al. (2010). *Advice interference* occurred between two competing aspect advices, while *weaving interference* occurred between the class code and the advice. Introduction interference, while possible in any AOP system, would only

occur if certain language-specific constructs exist in the source-code (e.g., the `declare` operation in AspectJ). Introduction interference related closely to structural interference, defined in the AWP taxonomy by Tian, Cooper, and Zhang (2010), and the CSS portion of the metric model by Bernardi and Di Lucca (2010). Advice and weaving interference fit into the AWP taxonomy's contextual and control flow interference (Tian, Cooper, & Zhang, 2010) and the metric model's CCF (Bernardi & Di Lucca, 2010). Contextual changes may have caused advice interference, while weaving interference may have caused control flow changes. Certain AOP metrics mapped directly to Tian, Cooper, and Zhang's (2010) AWP taxonomy, such as the coupling on advice execution (CAE) metric. The CAE metric measured the number of aspects advising a module (aspect and weaving interference), aspects declaring constructions on a module, and aspects defining inter-type declarations for a method (Ceccato & Tonella, 2004; Piveta et al., 2012). Given that advice and weaving interference depended on interactions between aspects advices and class methods, an analysis of a program's woven bytecode was performed in this work to reveal areas with a high potential for interference problems.

Detecting Interference at Design Time

The ultimate goal of properly defining aspect interference was its detection and elimination. One approach to detecting aspect interference in the literature was identifying the interference early in the software development life cycle. Chen, Ye, and Ding (2010) used formal induction methods to analyze and detect aspect interference problems at design-time. Basing their formal notation on the designs in *Unifying Theories of Programming* (Hoare, 1998), the authors described how to detect both weaving and advice interference. Using this approach, the authors showed formal definitions of the

design allowed for reasoning that both detected and solved interference in AOP systems, aside from behavior or model based techniques. Therefore, the authors concluded that this method would easily integrate into the AOP design process by adding annotations to design documents with function and advice definitions. (Chen et al., 2010)

Disenfeld and Katz (2012) extended formal aspect specifications by adding assumptions and guarantees to each aspect definition. When designated properly, the authors claimed that these additional specifications would allow proof of non-interference. The authors used the concept of joint weaving, which caused the weaving of all aspects at the same time without restrictions. For non-interference to hold, every pair of aspects would need to meet two rules. First, when weaving an aspect A into a system, the result of weaving needed to preserve the assumption of another aspect B. Second, when weaving an aspect B into a system, the result of the weaving needed to preserve the guarantee of the other aspect A. Using these definitions, the authors introduced and proved the soundness of a formal verification technique that guaranteed non-interference. The authors concluded that this approach would uncover aspect interference, dependencies, and cooperation that could exist in a system under development and would improve the quality of the system early in the process. (Disenfeld & Katz, 2012)

Disenfeld and Katz (2013) extended their previous work to include event and aspect verification. Using the same assumption and guarantee definitions on aspects as Disenfeld and Katz (2012), the authors noted that the verification of aspects alone was not enough to ensure non-interference. Disenfeld and Katz (2013) extended the definition of non-interference by including three conditions that must hold. First, verification of an aspect A assumed that every other aspect satisfied the aspects internal assumptions.

Second, an aspect A preserved both the assumption and guarantee of every other aspect. Third, any aspect activated within aspect A satisfied the aspect's internal assumptions. The authors defined two verification algorithms designed to assist with these definitions. The first automatically identified a weak assumption of an aspect with respect to events, while the second helped to find abstractions that would aid in stating event specifications. Both algorithms provided a means for verifying aspects, but also became part of an iterative process called CEGAR—counterexample guided abstraction refinement. The authors showed that this process was useful for finding aspect interference for both aspects within aspects as well as across pairs of aspects. (Disenfeld & Katz, 2013)

Other studies have employed a formal language definition of a system's static structure to analyze and detect interference. Hannousse, Douence, and Ardourel (2011) reviewed the definition and detection of aspect interference in component-based software engineering. The authors used an architectural description language (ADL) to describe the properties of both components and aspects in a component diagram. From this, the authors presented rules that converted ADL statements to networks of automata, and input these automata into a system called UPPAAL. UPPAAL provided a simulator that would allow programmers to examine system behaviors in depth. The authors used a motivating example that defined the component-based architecture of an airport wireless access system. Using this example, the authors showed that system designers could detect interference by analyzing the ADL description of a system with formal descriptions of system properties that must hold. The research concluded that programmers could use this technique to show that two aspects do not interfere if the base system was well defined and the weaving was correct with regard to the base system. However, to correct

a detected interference, the authors noted that developers would need to add some type of composition operator between interacting aspects. (Hannousse et al., 2011)

Design-time interference detection techniques attempted to guarantee that interference did not exist in the AOP system under development. Existing design-time techniques required formal definition by a developer (Chen et al., 2010; Disenfeld & Katz, 2012, 2013; Hannousse et al., 2011). Chen et al. (2010) and Disenfeld and Katz (2012, 2013) required annotations of design documents with formal aspect definitions. Hannousse et al. (2011) required the formal definition of the entire system, including the properties it should possess. Although these techniques detected advice and weaving interference early in the development, they added a layer of complexity that could introduce other unintentional problems. Such techniques would only be as good as the formal definitions specified by the designers. Therefore, designers and programmers needed either a development tool that made defining these specifications an easy process, or a technique that required no formal specifications to detect potential interference. This dissertation introduced a detection technique that required no formal specifications beyond the code itself. As stated by Hannousse et al. (2011), however, preventing the interference would need additional programming constructs to ensure appropriate behavior.

Detecting Interference at Implementation Time

While design-time techniques provided ways of annotating design documents to ensure non-interference, they did not account for decisions made by the programmer during the implementation. Even the smallest programming decisions could introduce interference into a system. Because of this, some research attempted to detect aspect

interference during the implementation phase of development. D'Ursi, Cavallaro, and Monga (2007) developed XCutter, a system that analyzed slices of woven bytecode. Program slicing was an existing method that defined sets of instructions that influenced some specified criteria. Object-oriented slicing techniques were not adequate for D'Ursi, Cavallaro, and Monga (2007) because slicing did not account for aspect weaving. Therefore, the authors suggested a four-step process for AOP slicing analysis. First, the process compiled and wove Java and AspectJ source code into bytecode using an AspectJ compiler. Next, the process applied preliminary analyses and existing slicing algorithms to the bytecode. The process then obtained a slice using these analyses. Finally, the bytecode in the slice mapped back to the original source code. The authors used the XCutter tool to study aspect interference within a motivating example. They noted that static analysis of the bytecode could pinpoint some of the problems associated with woven code. However, the authors also noted that simple slicing was not sufficient for finding potential interference because weaving always resulted in overlapping slices. In addition, slicing showed problems involving scope precision in the code that would require a slicer to track copies of each method to avoid false or repeated dependencies. Therefore, the authors rejected slicing as a technique for locating aspect interference. (D'Ursi et al., 2007)

Providing both design and implementation-time interference detection would offer a comprehensive solution. The implementation-time technique of D'Ursi et al. (2007) showed that static bytecode analysis was a viable option for detecting interference, but another analysis method was necessary because of the inadequacies found by using program slicing. Therefore, the current research proposed an

implementation-time technique that statically analyzed woven bytecode using clustering analysis. Clustering analysis did not allow overlapping groups. In addition, using vector state models in the clustering algorithm provided a more comprehensive view of the data than simple slicing, and produced favorable results.

Detecting Interference at Execution Time

Following the development phase of an AOP system, interference could still occur in the released product. Therefore, other researchers attempted to both detect and prevent interference when the code executed. Marot and Wuyts (2009) studied the problem of runtime interferences between advices. The authors noted that the existing solution of adding aspect precedence annotations in the code required global awareness of the precedence—breaking the separation of concerns by introducing dependencies that should not exist. Marot and Wuyts (2009) introduced an annotation language called *compositional intentions* to avoid using aspect precedence declarations and the problem of global aspect awareness. The language explicitly described both advice behavior and intention type. *Advice behavior* descriptions included a logically composed list of actions that an advice could perform, while *intention types* described the ways in which advice behavior should occur. If a violation of the compositional intention occurred at runtime, the system would produce an exception with a detailed explanation of the interference. The authors provided a small example that included no empirical results, but indicated future work would involve an implementation of compositional intentions using dynamic aspect scheduling. (Marot & Wuyts, 2009)

Marot and Wuyts (2010) extended their previous work (Marot & Wuyts, 2009) by exploring the problem of invasive aspect composition caused by aspect precedence

definitions using motivating examples. The authors showed that adding new aspects to an AOP system could require both pointcut and advice modifications to manage possible interactions, breaking the concept of aspect independence. As a result, the authors investigated the possibility of using aspects that compose other aspects to allow both foreign aspect modifications as well as precedence management. Marot and Wuyts (2010) proposed OARTA, an extention of AspectJ that added advice naming, advice patterns, foreign pointcut modification, user-defined instantiation policies, and altered AspectJ's `declare precedence` and `adviceexecution` constructs. The authors used the new OARTA language to revisit the motivating example, showing the required code alterations. (Marot & Wuyts, 2010)

Lauret, Fabre, and Waeselynck (2011) studied detection of both data- and control-flow interference at execution time. They described data flow interferences that included Change Before (CB) interference, when an aspect would access a base-code variable that an interfering aspect updated, and Change After (CA) interference, occurring when an aspect would access a base-code variable that competing aspects would later update. Control-flow interferences included Invalidation Before (IB) interference, occurring when a previous aspect has altered the join point so the competing aspect could not execute, and Invalidation After (IA) interference, occurring when an aspect would alter a join point that a competing aspect previously executed. The authors noted that any real-time detection scheme would need to watch six observation points along the execution path. However, they concluded that the most common language used for AOP—AspectJ—only provided for observation at three of these points. Groovy, on the other hand, provided for all six observation points. (Lauret et al., 2011)

To overcome the observation point problem in AspectJ, Lauret, Waeselnyck, and Fabre (2012) stated that the language must allow the definition of precedence at the advice level. To accomplish this, the authors used AIRIA, a system characterized by advices that compose other advices based on specific definitions of when and in what order the advices could execute. AIRIA was similar to OARTA, the language extensions developed by Marot and Wuyts (2010). AIRIA provided a layer atop AspectJ that allowed the authors to see all six observation points defined by Lauret et al. (2011). Thus, the system developed by Lauret et al. (2012) could use the defined advice precedence to determine interference problems at execution time and prevent them through assertions. The authors showed the feasibility of this solution with several examples, and concluded that the technique could detect both data- and control-flow interference problems at execution time. (Lauret et al., 2012)

Taking a different approach to execution-time detection, Figueroa (2013) and Figueroa, Tabareau, and Tanter (2013) described how to control aspect interference using monads and membranes in the Haskell programming language. Monads would allow programmers to chain structures together on a stack, while membranes would produce join points that only registered aspects could access. Figueroa, et al. (2013) defined ways to control aspect code interaction using control flow advice and non-interference advice on the monadic stack. Using membranes, the authors enforced non-interference in both control flow and data flow using the language's type system rather than relying on external analysis tools. The authors concluded that this system was a straightforward method to specifying and enforcing allowed interactions between aspects, and between aspects and system components. (Figueroa, 2013; Figueroa et al., 2013)

Execution-time techniques attempted to stop interference detected while the program ran. Marot and Wuyts (2009, 2010) noted that aspect precedence definitions challenged some of the fundamental intentions of aspect-oriented programming, and produced a system that combatted these issues. Lauret et al. (2011) showed that AspectJ was limited because it did not allow a run-time detection system to observe all of the locations along the execution path required to ensure non-interference. To overcome this problem, Lauret et al. (2012) provided a way to define precedence within the code at the advice level and use assertions to ensure non-interference in an executing program. Similarly, Figueroa (2013) and Figueroa et al. (2013) relied on Haskell's type system assertions to ensure non-interference. These techniques ensured that the program flow was interrupted when an assertion failed due to detected interference. Unfortunately, each of these techniques required very specific programming methodologies that would prove difficult for practitioners that lacked a good understanding of the entire system. While this dissertation did not require specific programming techniques for interference detection, it defined a way to help programmers find areas within the code that required interference prevention techniques.

Analysis

The goal of aspect interference detection has been to prevent bad effects within a system. As the definition of aspect interference became more refined, detection techniques opposed each other. Design-time techniques worked to ensure that an AOP system had no interference problems. Even with an interference-free design, programmers could introduce interference at implementation time. Implementation-time techniques located aspect interference introduced during the programming phase.

Execution-time techniques attempted to guarantee that a program ran without interference problems. Unfortunately, ensuring that a program was free of interference often required a program designer to use specific techniques that involved a full understanding of the program under development. To gain such understanding of a system, programmers needed assistance in finding potential advice and weaving interference in a program during the implementation phase. Thus, this dissertation's implementation-time interference detection technique gave a new way for the programmer to identify locations that required more specialized handling.

Aspect-Oriented Metrics

The Emergence of AOP Coupling Metrics

The use of metrics during software development allowed programmers to assess a product before completing the implementation. While some aspect-oriented metrics grew from object-oriented metrics, new aspect-oriented metrics emerged to describe the characteristics of AOP systems. One of the most striking features of AOP was the coupling it introduced between aspects and class code. Because this coupling could cause interference, well-defined aspect coupling metrics became essential for locating interference within a program. The evolution of AOP metrics showed a systematic refinement of aspect coupling metrics over time, beginning with a study of coupling from aspects to classes and later developing metrics that described other types of aspect coupling.

Zhao (2004) provided one of the earliest studies of aspect-oriented coupling metrics. To understand the types of metrics required, the author began by analyzing the types of coupling that existed within an AOP system. Zhao (2004) formally described

Table 1. Aspect Coupling Metric Development Introduced by Zhao (2004)

Dependency	Sub-Dependency	Metric	Description
Attribute-Class	Attribute-Class	Attribute-Class	The number of dependencies between attributes of aspect a and some set of classes C.
Module-Class	Advice-Class	Advice-Class	The number of dependencies between advices of aspect a and some set of classes C.
	Intertype-Class	Intertype-Class	The number of dependencies between intertype declarations of aspect a and some set of classes C.
	Method-Class	Method-Class	The number of dependencies between methods of aspect a and some set of classes C.
	Pointcut-Class	Pointcut-Class	The number of dependencies between pointcuts of aspect a and some set of classes C.
Module-Method	Advice-Method	Advice-Method	The number of dependencies between advices of aspect a and methods of the set of classes C.
	Intertype-Method	Intertype-Method	The number of dependencies between intertype declarations of aspect a and methods in some set of classes C.
	Method-Method	Method-Method	The number of dependencies between methods of aspect a and methods in some set of classes C.
	Pointcut-Method	Pointcut-Method	The number of dependencies between pointcuts of aspect a and methods in some set of classes C.
Aspect-Inheritance	Aspect-Inheritance	Aspect-Inheritance	The number of dependencies between aspect a and all ancestors of aspect a.

attribute-class dependence, module-class dependence, module-method dependence, and aspect-inheritance dependence. From these couplings, Zhao (2004) developed and formally described ten measurements to assess each dependency. Table 1 details these measurements. Zhao (2004) noted that the coupling studied was from aspects and their components to classes and their components. This only described the fan-out from aspect perspective and failed to acknowledge the fan-in from the class code perspective. The study also failed to address aspect-aspect coupling. The author presented no empirical analysis of the suggested metric framework. (Zhao, 2004)

Ceccato and Tonella (2004) provided another early set of coupling metrics for aspect-oriented programming. This study updated several classic OO metrics for use in AOP, including Weighted Operations in Module (WOM), Depth of the Inheritance Tree (DIT), Number of Children (NOC), Response for a Module (RFM), and Lack of Cohesion in Operations (LCO) (Chidamber & Kemerer, 1994). In addition, the authors identified new metrics specific to AOP, including Coupling on Advice Execution (CAE),

Coupling on Intercepted Modules (CIM), Coupling on Method Call (CMC), Coupling on Field Access (CFA), and Crosscutting Degree of an Aspect (CDA). Table 2 defines each of these metrics. CMC and CFA were extensions of the OOP Coupling Between Methods (CBM) metric (Chidamber & Kemerer, 1994). CAE, CIM, CMC, CFA, RFM, and CDA provided a picture of coupling at various granularities. The authors noted that CIM, CMC and CAE correspond to pointcut-class, pointcut-method, and method-method dependencies described by Zhao (2004), but CDA did not map to any of Zhao's (2004) dependency measures. (Ceccato & Tonella, 2004)

Table 2. Aspect-Oriented Metrics Introduced by Ceccato and Tonella (2004)

Weighted Operations in Module (WOM)	The number of advices or methods in a given aspect or class.
Depth of the Inheritance Tree (DIT)	The length of the longest path from a given aspect or class to the root.
Number of Children (NOC)	The number of immediate sub-classes or sub-aspects of a given aspect or class.
Response for a Module (RFM)	The number of methods or advices potentially executed in response to a given class or aspect.
Lack of Cohesion in Operations (LCO)	Pairs of advices or methods that work on different class fields minus pairs of advices or methods that work on common fields.
Coupling on Advice Execution (CAE)	The number of aspects that contain advices triggered by the advices or methods of a given class or aspect.
Coupling on Intercepted Modules (CIM)	The number of classes, aspects, or interfaces named in pointcuts that belong to a given aspect.
Coupling on Method Call (CMC)	The number of classes, aspects, or interfaces declaring methods potentially called by a class or aspect.
Coupling on Field Access (CFA)	The number of classes, aspects, or interfaces with fields that are accessed by a given class or aspect.
Crosscutting Degree of an Aspect (CDA)	The number of classes or aspects affected by the pointcuts and by the introductions in a given aspect.

Ceccato and Tonella (2004) developed a metrics tool that analyzed an AOP program through a process of reverse-engineering intertype declarations, method calls and field accesses, and pointcuts. The authors suggested what high or low readings for each metric indicated, but did not show empirical results. Instead, the authors simply applied their metric tool to Java and AspectJ implementations of the Observer design pattern and showed the resulting metric values. (Ceccato & Tonella, 2004)

Kumar, Kumar, and Grover (2009) extended the work of Ceccato and Tonnella

(2004), Zhao (2004), and others by introducing a new coupling metrics framework for AOP. Their research specified 17 types of connections that exist in an AOP system between attributes, operations, or components. From these 17 connection types, the authors developed six new metrics, described in Table 3: Coupling on Attribute Type (CoAT), Coupling on Parameter Type (CoPT), Coupling on Attribute Reference (CoAR), Coupling on Object Invocation (CoOI), Coupling on Inheritance (CoI), and Coupling on High-Level Association (CoHA). The authors acknowledged that these metrics were simply a framework, and lacked empirical evaluation. No other studies appear to either verify or utilize these metrics. (Kumar et al., 2009)

Table 3. Aspect Coupling Metrics Introduced by Kumar et al. (2009)

Coupling on Attribute Type (CoAT)	The total number of attributes of a component that interact with another component.
Coupling on Parameter Type (CoPT)	The total number of operations from one component coupled to another through parameter types, local variables, and return types.
Coupling on Attribute Reference (CoAR)	The number of operations from one component that references an attribute of another component either statically, by inheritance, or dynamically.
Coupling on Object Invocation (CoOI)	The number of operations from one component that invokes an operation of another component explicitly or implicitly.
Coupling on Inheritance (CoI)	The number of ancestors of a given component.
Coupling on High-Level Association (CoHA)	The number of high-level relationships between components (e.g., "uses" or "consists").

Empirical Studies of AOP Metrics

Burrows et al. (2010) noted the lack of empirical analysis of AOP metrics in the literature for determining potential faults in a system. The authors analyzed the AOP metrics discussed by Ceccato and Tonella (2004), while adding a new metric called Base-Aspect Coupling (BAC) to account for coupling between the base and aspect code. Burrows et al. (2010) conducted experimental evaluation on iBATIS, an object-relational mapping tool, using four releases with known faults. For analysis, the authors applied Spearman's Rank correlation coefficients using the number of known faults and the fault density. The authors identified three groupings: Group A (quantification metrics), which

included CDA and BAC, Group B (fine-grained metrics), which included CMC, CFA, CBM, and WOM, and Group C (coupling granularity and CAE), which included LCO, RFM, CAE, DIT, and NOC. Results showed that Group A was the most significant for detecting faults, with fault-count coefficients of 0.30562110 ($p = 0.00000017$) for CDA and 0.26968580 ($p = 0.04646000$) for BAC and fault density coefficients of 0.26918280 ($p = 0.04689000$) for CDA and 0.26854670 ($p = 0.04743000$) for BAC. Group B also showed significant results in CMC, CFA, and CBM, though the finer-grained CMC and CFA metrics showed stronger fault-detection capabilities than the more coarsely grained CBM metric. Group C metrics did not show significant results. When comparing CAE and BAC, the authors noted that both metrics associate base classes with aspect code, but CAE did not show significant results due to the metrics' differing dimensions. These results suggested that classic AOP metrics might require adjustments to make them more effective for finding potential faults in a system. (Burrows et al., 2010)

Piveta et al. (2012) studied AOP metrics LOCC, WOM, DIT, NOC, CDA, and CAE more rigorously than previous research. They first created formal definitions of each metric, and then applied them to ten different AOP projects. The authors defined six properties for evaluating each metric: non-coarseness, non-uniqueness, having important design details, monotonicity, non-equivalence of interaction, and interaction increases complexity. Non-coarseness described a metric's ability to have different values for different modules. Non-uniqueness was the property that the metric could be the same for two different modules. Having different design details meant that a metric could have different values for different designs. Monotonicity was the property that the value of the

metric for composed objects was never less than the value for their individual components. Non-equivalence of interaction meant that the composition of two modules A and B could result in a different metric value than the composition of modules A and C. Interaction increases complexity meant that the metric value could increase when composing two modules. (Piveta et al., 2012)

Piveta et al. (2012) showed that LOCC and WOM met all six properties. DIT and NOC failed to satisfy the interaction increases complexity property, and only satisfied the monotonicity property under certain conditions. CDA and CAE satisfied all except the interaction increases complexity property. The authors stated that high values for LOCC indicated high complexity in the module, while low values indicated potentially unnecessary modules. High values of WOM pinpointed unnecessary class couplings and potential for aspect refactoring, while low values of WOM indicated the need to combine or remove smaller classes or aspects. High DIT showed the possibility of removing unnecessary levels of inheritance, while low DIT values were normal, except when a complex class could benefit from inheritance. High NOC values showed that a class or aspect was highly reused within an inheritance structure, while low NOC was normal. High CDA values showed that an aspect was widely utilized within the class code, while low CDA could indicate that the aspect in not needed. High CAE values could indicate the possibility of aspect interaction, while low CAE values were common. Correlation analysis indicated that a strong correlation existed between LOCC and WOM ($r^2 = 0.76$) and a smaller correlation between LOCC and CDA ($r^2 = 0.26$) and WOM and CDA ($r^2 = 0.16$). This indicated that an aspect's size related to its crosscutting performance. The authors concluded that each of these metrics was useful for finding

specific issues, and the correlations indicated provided meaningful information about a program. (Piveta et al., 2012)

Analysis

AOP metrics grew largely from the fact that coupling was inherent to the aspect-oriented paradigm. Although Zhao (2004) provided the earliest introduction of aspect coupling measurements, metrics introduced by Ceccato and Tonnella (2004), including the adapted OO metrics of Chidamber and Kemerer (1994), have been studied and used throughout the literature (Burrows et al., 2010; Kumar et al., 2009; Piveta et al., 2012). The empirical results of Burrows et al. (2010) showed a clear advantage for quantification metrics, but also noted an advantage for fine-grained metrics CMC, CFA, and CBM. These results indicated a stronger result for CMC and CFA, which were finer-grained versions of CBM. Studies involving clustering analysis (mentioned in the next section) have used fine-grained metrics to analyze method interactivity. Because the goal of this study was to find potential advice and weaving interference, it required a fine-grained metric to describe coupling at the advice and method granularity. The most granular aspect coupling metrics described in the literature addressed coupling between advices or methods as related to aspects or classes. Although Zhao (2004) described method-method dependencies, the dependencies were enumerated from an aspect's method to a class's method. This study required metrics that enumerated the relationships among aspects and classes at the advice-method, advice-advice, and method-method granularities. Therefore, this study introduced two new coupling metrics to account for each of these couplings: interference potential (IP) and interference coupling potential (ICP). Supporting the results of Burrows et al. (2010), these new fine-grained metrics

provided satisfactory outcomes when used in clustering analysis.

Clustering Analysis

Based on existing aspect interference research, programmers needed to either design a system based on specific formal definitions, or manually locate potential aspect interference and refactor the system to eliminate the interference. However, manually locating potential aspect interference required programmers to understand how all aspects interacted with each other and with the base program during the development process—a task that became increasingly complex as system size increased. Some research attempted to increase understanding of a system to locate areas of a program that were candidates for refactoring. One method used in software development for finding refactoring potential was clustering analysis, which looked for specific trends within source code.

Clustering in OOP

I. G. Czibula and Şerban (2006) used *k*-means clustering to help identify ways to refactor existing OO code. The clustering process began by analyzing the code to determine relevant components and their existing relationships. Next, the process re-grouped components using a *k*-means clustering algorithm named *kRED* (*k*-means for Refactorings Determination). This algorithm set the initial number of clusters to the number of classes in the system. Then, it chose the classes themselves as the initial centroids. Next, the algorithm repeatedly calculated distances so that each object was a member of the closest cluster, and stopped when two iterations remained unchanged or when the number of steps exceeded the maximum iterations. The algorithm produced an improved system structure. Finally, the authors compared the newly created system structure to the original structure to determine refactorings. The authors implemented the

detection of move method, move attribute, and inline class refactorings, but noted that the algorithm could accommodate other types of refactoring as well. I. G. Czibula and Şerban (2006) used JHotDraw for experimental evaluation, and defined two new metrics: accuracy of a refactoring technique (ACC) and precision of a refactoring technique ($PREC$). When applied to JHotDraw, the clustering method showed an ACC value of 0.9829 and a $PREC$ value of 0.9956. The algorithm found six misplaced methods within the source code. When compared to a previous method, this algorithm obtained a higher precision, but the authors could not determine an accuracy comparison. (I. G. Czibula & Şerban, 2006)

Hussain and Rahman (2013) used a hierarchical agglomerative clustering technique to support software restructuring. The authors' approach first analyzed a function to determine an entity-attribute matrix. Entities in the model were functional lines of code, while attributes came from elements that an entity used. Next, the algorithm calculated the similarity or dissimilarity between entities, and then performed clustering using a new hierarchical technique named (*k*,*w*)-Core Clustering ((*k*,*w*)-CC). The authors defined (*k*,*w*)-CC by relying on graph theory to translate the similarity matrix into graphical clusters. The algorithm first decomposed the system into (*k*,*w*)-cores, where *w* is the edge weight. Next, (*k*,*w*)-CC selected cores based on the new metric called relatedness, and finally generated the clustering tree. Experimental evaluation compared the restructuring of systems using the SLINK, CLINK, WPGMA, A-KNN, and (*k*,*w*)-CC algorithms. Results showed that (*k*,*w*)-CC produced smaller numbers of cut-points and bad clusters than the other methods by discarding both redundant and inferior-quality results. This indicated that (*k*,*w*)-CC produced larger clusters by looking at structural

properties other than cluster/entity similarity. (Hussain & Rahman, 2013)

Other authors have used clustering analysis to verify software design principles. Yu and Ramaswamy (2007) noted the hierarchical nature of software systems and applied hierarchical clustering techniques to determine the level of modularity, hierarchy, and interaction locality in an OO design. The authors defined *interaction frequency* as the degree of interactivity between two system components. They represented this idea in an $n \times n$ interaction matrix in which the intersection of two components listed the interaction frequency between them. The clustering algorithm began by adding each component in the design to its own cluster and then merging pairs of clusters with the highest interaction frequency. Finally, the algorithm found the interaction frequency between the new cluster and the old clusters. The authors used KWIC (Key Word In Context) as an experimental case study. Results were limited with this case study because the only interaction frequency used was parameter coupling. The authors recognized that this limited the production of the interaction matrix, and ignored interactions that were more complex. (Yu & Ramaswamy, 2007)

Yu and Ramaswamy (2009) extended their work by adding the adapted concepts of spatial and temporal distance using hierarchical clusters of software components. Spatial distance represented the distance between two components in the hierarchical cluster, while temporal distance represented component relationships based on revision histories. The authors mapped spatial distance and temporal distance into two $n \times n$ matrices and used the Mantel test (Mantel, 1967) to correlate the two. Yu and Ramaswamy (2009) evaluated their method using six open-source Apache projects: Ant, DB, HTTP, Lenya, Tomcat, and XML. Following the analysis, the authors concluded that

the spatial distance metric was proportional to complexity, while the temporal distance metric related to the logical dependencies between code components. The authors used the correlation between these metrics to represent the software's overall quality, but acknowledged that the approach ignored several factors—including the modularity type and the architecture. In addition, the authors noted that the *p*-values computed by the Mantel tests seemed an inconclusive means of determining dependency locality, but stated that Lenya ($p = 0.2$) likely had the poorest dependency locality of the six systems evaluated. (Yu & Ramaswamy, 2009)

Source code analysis was highly relevant to this dissertation. Finding areas that exhibited aspect interference was quite similar to locating refactoring potential. Determining refactoring potential (I. G. Czibula & Şerban, 2006; Hussain & Rahman, 2013) in an AOP program required analysis of component interaction within the source code, similar to software verification techniques in the literature (Yu & Ramaswamy, 2007, 2009). Adapting clustering techniques to AOP has provided a way to pinpoint potential advice and weaving interference within an AOP program.

Clustering in AOP

No known studies have used clustering to locate potential aspect interference in existing code. However, clustering research in AOP has focused on finding crosscutting concerns within existing class code—a technique called *aspect mining*. Shepherd and Pollock (2005) provided one of the earliest studies in aspect mining. The proof-of-concept study used hierarchical agglomerative clustering to locate potential crosscutting concerns. The clustering model defined one cluster per method, and recursively grouped clusters until their distance was less than a predefined threshold. To determine the

distance between two methods, Shepherd and Pollock (2005) used a simple function based on the lengths of method names. This clustering also presented a basic hierarchical visualization of each cluster, as the authors stored clusters as trees. The authors validated their approach using JHotDraw as a case study. Through this study, the authors observed and explained three distinct categories of crosscutting concerns: those whose interfaces were consistently implemented, those whose interfaces were inconsistently implemented, and those with no explicit interface. Shepherd and Pollock (2005) contended that this method would allow programmers to determine clusters with crosscutting concerns versus simple code duplication. Their stated advantages suggested that the approach was extensible because the distance function was easily changeable, was powerful but required no extra computation, and was a first step toward combining mining and viewing research. (Shepherd & Pollock, 2005)

Moldovan and Şerban (2006) were the first to use vector-space models in aspect mining, allowing a more detailed model than a simple metric could provide. The authors used two vector space models: $\mathcal{M}_1$ used vectors for each method defined by $\{FIV, CC\}$, where FIV was the fan-in-value and CC was the calling classes, and $\mathcal{M}_2$ used vectors defined by $\{FIV, B_1, B_2, \dots, B_{l-1}\}$, where FIV was the fan-in-value and the value of B_k $(1 \leq k \leq l-1)$ was 1 if the method was called from the corresponding class, or 0 otherwise. The authors used Euclidean distance between vectors to determine distance, and the reciprocal of the Euclidean distance to determine similarity. The authors accomplished clustering through adaptations of the *k*-means and hierarchical agglomerative clustering algorithms. Evaluating the approach, the authors applied the algorithms to Theatre, the Laffra implementation of the Dijkstra algorithm, and

JHotDraw. The hierarchical agglomerative clustering algorithm and the *k*-means clustering algorithm both showed similar clustering results in the experimentation. The cases studied revealed that the first two clusters contained nearly the same methods regardless of the clustering algorithm, and that these methods implemented crosscutting concerns. (Moldovan & Şerban, 2006)

To extend these results, Şerban and Moldovan (2006) presented a novel *k*-means clustering algorithm for aspect mining named *kAM*. This algorithm used a heuristic method to determine the number of clusters and the initial centroids. To calculate centroids, the algorithm chose the most distant method as the first centroid. Next, it repeatedly found the minimum distance (*dmin*) from each remaining method and the initial centroid. The next centroid chosen had the maximum *dmin* value. After finding the initial *k* centroids, the algorithm behaved like the classical *k*-means algorithm. This study used the same two vector models as Moldovan and Şerban (2006). The authors proposed four metrics to evaluate the results: intra-cluster distance in a partition, inter-cluster distance in a partition, precision of a clustering based aspect mining technique, and percentage of analyzed methods for a partition. Experimental evaluation applied *kAM* to the Laffra implementation of the Dijkstra algorithm and to JHotDraw. Results were mixed because the clustering algorithm favored both vector space models in different experiments. This led the authors to conclude that improvements in vector space models would be required. (Şerban & Moldovan, 2006)

Tribbey and Mitropoulos (2012) noted that vector spaces often used aggregated values such as FIV as components. To improve such vector space models, the authors introduced a matrix-based vector model (M_{FIV}) organized into an $n \times n$ bitmap P, where

n was the number of modules. If module m_i called module m_j, the value at location p_{ij} in matrix P became 1, or remained 0 otherwise. When summed, each matrix row was equivalent to the FIV, yet the matrix preserved all relationships. For comparison, the authors created two other vector space models based on fan-out-value (M_{FOV}) and a composite model based on the first two (M_{COM}). Based on the studies of Moldovan and Şerban (2006) and Şerban and Moldovan (2006), the authors chose the *k*-means algorithm to perform partitioning. The SD index determined the value for K that fed into the algorithm. Using JHotDraw, the authors applied partitioning, and reduced the number of dimensions using Principle Component Analysis. While the new model proved viable, the results showed mixed results across the three vector space models. The authors concluded that issues existed with the determination of crosscutting concerns and the measurements used. Tribbey and Mitropoulos (2012) noted that future work in evaluating mining algorithms and test data was necessary. (Tribbey & Mitropoulos, 2012)

G. Czibula et al. (2009) introduced a partitional clustering algorithm for identifying crosscutting concerns called *PACO*. The *PACO* process began by analyzing the source code to identify all classes, methods, and the relationships between them. Each method initially became its own cluster. Next, the algorithm chose the most distant method as the first medoid (centroid). The algorithm then recursively found the next medoids by finding the points that maximize the minimum distance to the methods. This method was similar to the heuristic method mentioned by Şerban and Moldovan (2006). *PACO* continued to refine the clusters based on the original heuristic. It recalculated each cluster and then recalculated the medoid of each cluster repeatedly until the medoid remained unchanged. The authors evaluated the process using JHotDraw, which showed

an improved dispersion of crosscutting concerns (DISP) when compared to *kAM* (Şerban & Moldovan, 2006). Therefore, the authors concluded that *PACO* provided better clustering of crosscutting concerns than *kAM*. (G. Czibula et al., 2009)

Rand McFadden and Mitropoulos (2012) were the first to apply model-based clustering techniques to aspect mining instead of heuristic methods. The authors defined six vector-space models: M_1 (fanIn_numCallers), M_2 (fanIn_hasMethod), M_3 (sigTokens), M_4 (fanIn_sigTokens), M_5 (fanIn_numCallers_sigTokens), and M_6 (fanIn_numCallers_hasMethod_sigTokens). M_4 combined *FIV* with M_3, M_5 combined M_1 and M_3, and M_6 combined M_1, M_2, and M_3. Six clustering algorithms provided results: partitioning methods *k*-means (KMH), *k*-means with random initial centroids (KMR), and hierarchical agglomerative clustering (AGN), and model-based methods MCL, hierarchical agglomerative clustering (HC), and presence-absence clustering (PRAC). Evaluating the algorithms and vector space models on JHotDraw, results showed that the best overall result was HC, using M_5. These experiments showed that model-based methods had improvement with respect to scattering concerns across multiple clusters, and with respect to partitioning. Further, the authors found that a combination of previously defined vectors and newly defined vectors performed best across all methods. (Rand McFadden & Mitropoulos, 2012)

Analysis

Aspect mining research attempted to find ways to refactor code by determining where aspects existed within object-oriented source code. Similarly, the current research attempted to locate ways to refactor code by determining where potential aspect interference existed within aspect-oriented code. Since Shepherd and Pollock (2005),

mining research grew from simple metrics-based analysis to vector space analysis. Vector spaces introduced by Moldovan and Şerban (2006) have become the basis for other aspect mining studies. While findings in vector space models continued to be mixed (Şerban & Moldovan, 2006; Tribbey & Mitropoulos, 2012), they provided a more detailed clustering model than traditional metrics. For partitioning, most research has used variations of the *k*-means or hierarchical agglomerative clustering algorithms. Determining initial clusters heuristically (Şerban & Moldovan, 2006) or improving clusters heuristically (G. Czibula et al., 2009; Tribbey & Mitropoulos, 2012) have shown positive results. According to Rand McFadden and Mitropoulos (2012), model-based algorithms using hybrid vector spaces can result in further improvements.

These advances have proven that clustering is a viable means for determining potential for refactoring in an existing piece of software. However, the metrics used in these studies do not directly map to aspect interference. Most studies used a method's FIV as a component of the clustering model. FIV counted the number of methods that call the method. Tribbey and Mitropoulos (2012) noted that FIV would indicate code scattering, but not code tangling. *Code scattering* referred to a crosscutting concern that spread across a system, while *code tangling* referred to one crosscutting concern mixing with other concerns (Şerban & Moldovan, 2006). Tribbey and Mitropoulos (2012) also used FOV in a vector space model in their clustering assessment to ensure coverage of code tangling. Applying these ideas to aspect interference research, *interference scattering* would describe interference that spread across an AOP system, while *interference tangling* would refer to multiple interferences from a given object. In the case of aspect interference, the clustering models needed to account for both interference

scattering and interference tangling to ensure a full understanding of potential interference. Existing AOP metrics such as coupling on advice execution, and crosscutting degree of an aspect illustrated coupling at the aspect level (Ceccato & Tonella, 2004; Piveta et al., 2012). To describe potential advice and weaving interference in terms of interference scattering and interference tangling, finer-grained metrics were required. Therefore, extension of previous techniques to use the new IP and ICP metrics to account for method-method, advice-advice, and advice-method interaction was imperative to the current study.

Visualization

While clustering provided a good analysis tool when applied to existing code, programmers also needed to understand the results of the clustering before addressing interference in the code. One method that allowed programmers to understand these potential interactions better was visualization. Though the literature has shown the importance of visualization for design analysis in OOP, no studies have applied visualization techniques to aspect interference. They have, however, applied visualization in the context of clustering, and have considered combining the two an important next-step in research (Shepherd & Pollock, 2005).

Design Visualization Techniques in OOP

Like clustering, the goal of some visualization techniques was to improve the design of an existing piece of software by illustrating design problems. Lanza and Martinescu (2006) noted that developers should use metrics in conjunction with visualization techniques to assist in understanding complex designs. The authors defined a *polymetric view*, which displayed a set of metrics visually by utilizing a node's size,

color, position, and the edge's color and width. Using this polymetric view, the authors advocated the idea of *design harmony*. Design harmony specified that each system artifact should exhibit identity harmony (harmony with itself), collaboration harmony (harmony with its collaborators), and classification harmony (harmony with its ancestors and descendants). The authors noted that instances of disharmony would be detectable in system visualizations called class blueprints. Class blueprints represented the static class structure and focused on method calls, attribute access, and inheritance. Using specific detection strategies, the authors concluded that class blueprints highlighted disharmonies in a design, but became visually complex for large systems. (Lanza & Marinescu, 2006)

To overcome some of the complexity in existing visualization techniques, Wettel and Lanza (2008) extended the idea of polymetric views by describing a visualization metaphor called a *code city* that looked much like a three-dimensional city map. Each code city contained districts that represented packages, and buildings that represented classes. The sizes of each component came from specific metrics for that component, with the largest buildings representing the most impactful classes. The authors used detection strategies to color-code the classes in the code city based on their level of disharmony. Experimental evaluation applied the process to four Java programs: JDK, ArgoUML, Jmol, and iText. Results showed that, despite exhibiting an organized structure, JDK suffered from design disharmony, as many classes performed more than they should. iText showed disharmony scattered throughout due to its lack of organization. ArgoUML exhibited a variety of disharmonies, having classes with few attributes and many methods, and classes with little functionality. The authors concluded that this visualization technique produced false first impressions that one could eliminate

if using more detailed metrics. However, Wettel and Lanza (2008) contended that using more detailed metrics would reduce the human mind's ability to grasp the visualization. (Wettel & Lanza, 2008)

Clustering Visualization Techniques

Other authors have combined clustering and visualization techniques to show ways to refactor code. Dietrich, et al. (2008) used Java dependency graphs and located clusters within the graph based on these dependencies. To do this, the authors used the Girvan-Newman algorithm, which repeatedly removed edges with the highest betweenness from the dependency graph until reaching an acceptable clustering. The authors created a visual dependency graph using the Prefuse visualization toolkit and drew colored boxes around modules to represent clusters. They determined modularity in the visual graph by measuring the average number of packages per cluster and the average number of clusters per package. This allowed the developer to note potential changes that could increase the code's modularity. The authors tested the tool's scalability using several programs, including Xerces, Xalan, Commons-collections, the MySQL ConnectorJ JDBC driver, and a large piece of software supplied by New Zealand's Kiwiplan Company. Results showed that the tool was highly scalable to large projects when using a dual-core system with 2 GB of RAM. (Dietrich et al., 2008)

In a similar study, Cassell, Anslow, Groves, and Andreae (2011) developed ExtC Visualizer, a Java program that displayed relationships between classes in the form of dependency graphs. The tool allowed users to select either agglomerative clustering (which merged small clusters into larger ones) or divisive clustering (which split large clusters into smaller ones) to analyze the best results for identifying refactoring

opportunities. The authors examined approximately 100 classes from several different programs: Heritrix, Jena, JHotDraw, Weka, and their own ExtC product. Their observations led to three basic conclusions. First, clustering algorithms needed to consider domain knowledge to help refactor large classes, because the tool required programmers to view the clustering and make decisions about how to refactor the code. Second, the observations led to insights into how clustering algorithms behaved. The authors suggested that divisive clustering techniques were favored over agglomerative techniques, claiming that divisive clustering would be easier for programmers to understand because refactoring generally splits classes apart one by one. In addition, divisive clustering would require fewer processing steps than agglomerative clustering. Finally, the observations led the authors to suggest possible improvements to the visualizations. Extending ExtC to a scope lower than the class-level would require visualizations to display interclass relationships. (Cassell et al., 2011)

AOP Visualization

The work in AOP visualization has often surrounded the development of aspect debuggers. DeBorger, Lagaisse, and Joosen (2009) noted that AOP development tools lacked the ability to show point cuts and advices, and "concrete aspect-based abstractions" (p. 174). Additionally, DeBorger et al. (2009) stated that traceability from advice execution to the source code of such systems was difficult. Therefore, the authors developed a runtime visualization that had six main requirements: to allow inspection of applied advices, executing advices, past advices, the causal point cut, aspect instances, and the program's structure. To meet these requirements, the authors created a system established on the mirror-based reflective architecture for debugging systems. This

architecture provided a way to track a programs structure as well as the causality between aspects and the structure. The authors' AJDI system extended the Java Debugging Interface by adding mirrors for Aspect, Advice, Binding, AdviceApplication, JoinPoin, HookFrame (designed to trace join points and advices on the stack), and PastAdvice. The authors defined the Aspect Debugger (ADB), which implemented the extended architecture. To validate AJDI, the authors applied the ADB to both AspectJ ABC and JBoss AOP. The authors illustrated the capabilities of the AJDI system by introducing six bugs into the IconViewer application. The authors concluded that AJDI met all requirements, and left further validation and integration into Eclipse to future research. (DeBorger et al., 2009)

Fabry, Kellens, and Ducasse (2011) noted the difficulty of knowing both at which join points an aspect executed, and the order in which aspect code executed. The authors acknowledged the need for developer tools that clearly show the impact of aspects on the existing class code. Because of this, Fabry et al. (2011) developed AspectMaps—a visualization that illustrated join point shadows within a program. *Join point shadows* were locations in the code that became join points when the program executed. The visualization showed join point shadows from a coarser level to a more fine-grained level by allowing the user to zoom from a package-level display into a method- or advice-level display. Using AspectJ program spacewar as a motivating example, the authors conducted a small user study to determine the utility of the AspectMaps tool. While results showed a higher code comprehension over AspectJ Development Tools, the user study included a very small group of non-typical developers who performed non-standard tasks. (Fabry et al., 2011)

Yin, Bockisch, and Aksit (2012) noted that most bugs reported in AOP came from implicit invocation, making them difficult to detect and trace because code elements could be lost after compilation. Therefore, the authors proposed a fine-grained debugger for AspectJ to support a wider range of tasks than previous debuggers. The research defined ten tasks that an AOP debugger should perform: setting AO breakpoints, locating AO constructs, evaluating pointcut sub-expressions, flattening pointcut references, evaluating pattern sub-expressions, inspecting runtime values, inspecting AO-conforming stack traces, inspecting program compositions, inspecting precedence dependencies, and excluding and adding AO definitions (Yin et al., 2012, p. 62, Table 2). The authors defined the Advanced-Dispatching Debug Interface (ADDI) that used an intermediate representation of the program in XML, and added mirroring to JDI similar to De Borger et al. (2009). The user interface, integrated into Eclipse, allowed programmers a visual view of the system from the join point perspective. The authors concluded that the ADDI system performed all ten tasks, fully implementing six of them for the first time. (Yin et al., 2012)

Yin (2013) proposed a system that focused on visualizing advice interaction at join point shadows. Yin's (2013) approach required several components. First, the proposed system required an omniscient debugger to provide important information about specific join points in a program at runtime. To accomplish this, the author planned to use NOIRIn, "an execution environment that models advanced-dispatching (AD) as first class objects" (Yin, 2013, p. 29). Second, the visualization tool needed a query language for searching execution histories. To accomplish this, Yin (2013) suggested a graphical approach to reduce the complexity of textual queries and to eliminate the need to learn

new query languages. Third, the proposed tool needed an algorithm that would sift the execution history for which data to display graphically. Despite these plans, Yin (2013) left the implementation and experimentation to future research, concluding that this would be the first AOP visualization system to focus on changes at join points.

Analysis

Visual representations of a program have been important tools for a developer because they increased understanding of potential design problems. Giving a polymetric view of a design showed areas of a program that exhibited bad design practices (Lanza & Marinescu, 2006; Wettel & Lanza, 2008). Unfortunately, such polymetric views of a program only allowed for a small number of metrics before the visualization became too complex (e.g., three dimensions versus four dimensions). Therefore, other authors suggested clustering techniques to provide better inputs to complex visualizations. Using dependency graphs based on clustering results, research suggested that programmers could gauge a program's modularity and the need to refactor code (Cassell et al., 2011; Dietrich et al., 2008). This progression also illustrated that developing a better understanding of a system required detailed information, and mirrored the progression of clustering techniques from using simple metrics to more complex vector space models.

Determining the need to refactor AOP code to eliminate aspect interference was the core of the current study. Unfortunately, no studies applied clustering visualization to AOP code specifically for aspect interference. The study of Fabry et al. (2011) applied the idea of zooming in and out of an AOP visualization to show connections between different levels but did not discuss aspect interference. Yin (2013) seemed to confirm the idea of visualizing join point shadows, but did not offer an implemented solution. Other

studies in AOP visualization focused on providing debugging tools that ensured both visibility and traceability of execution paths (DeBorger et al., 2009; Yin et al., 2012). While such debugging tools were important to developers, they missed the opportunity to show potential refactorings. For example, one portion of the study by Yin et al. (2012) gave a way to visualize aspect precedence defined within the code (which could prevent aspect interference). Such a visualization was one-sided because it only showed the result of precedence definitions rather than pinpointing potential aspect interference. Thus, the current work extended the work of Fabry et al. (2011) and Yin (2013) into the area of aspect interference by providing a way to visualize potential areas of aspect interference within an AOP program.

Summary

Aspect-oriented research documented the existence of unwanted aspect interactions early in the paradigm (Douence et al., 2002). Researchers have since developed increasingly descriptive definitions of aspect interference. The more robust definitions of advice, weaving, and introduction interference (Tian, Cooper, Zhang, et al., 2010) gave a better foundation for recognizing instances of aspect interference within a program. Both the detection and prevention of aspect interference has become an important thread of research.

Detection of aspect interference occurred at three different points along a project's lifecycle. First, authors proposed the development of interference-free applications by creating interference-free designs. Design-time strategies required formal descriptions of the system that became more complex as system sizes increased, making them less effective for smaller-scale programming projects. Second, authors proposed the

detection of aspect interference during the implementation phase. The use of static code analysis provided an implementation-time technique, but few studies existed in that area. The use of program slicing as a static code analysis method was found to be a poor choice for locating interference (D'Ursi et al., 2007), and no other static code analysis interference detection studies existed. Finally, other researchers have studied the prevention of aspect interference problems at execution time. These techniques required programmers to state precedence rules explicitly that, when not followed, would cause exceptions within the code. Both design time and execution time techniques required a vast understanding of the program under development. Often, because of team development environments, developers only understood portions of a program without seeing the larger picture. Therefore, continuing the study of aspect interference via static code analysis was an essential next step in the evolution of the literature.

Clustering analysis (a static analysis technique) has proven effective for locating potential refactoring opportunities in object-oriented code. Other research used clustering to locate potential aspects in an existing OO system. Extending clustering analysis to existing AOP systems allowed the identification of aspect interference by reviewing the code statically. Unfortunately, existing AOP metrics in the literature failed to describe all possible interactions required to produce a proper clustering. Therefore, the introduction of new fine-grained AOP coupling metrics was required to perform clustering analysis. Because of these things, clustering analysis and finer-grained metrics provided worthwhile extensions to the aspect interference body of knowledge.

Visualizations have increased programmer understanding of both object-oriented and aspect-oriented programs. Some work in AOP visualization has involved the

development of debugger add-ons that assist the programmer in finding coding errors. While some components of these tools have included aspect interaction, none has focused exclusively on aspect interference. Therefore, using visualizations specifically for aspect interference was a meaningful addition to the body of research.

Chapter 3

Methodology

Overview

This study followed an experimental methodology, employing a novel approach for determining the potential impact of aspect interference within a program by using clustering analysis. Clustering results were validated using existing AOP systems to determine both the feasibility of the approach and scalability. The outcome of the clustering analysis provided an input for a clustering visualization to show potential aspect interference within the code.

The approach involved the following basic steps. A more-detailed explanation of each step, and the constructs used, follows this initial listing.

1. **Source Code Compilation:** The first step in the analysis converted the source code of an AspectJ system into woven Java bytecode by using the ajc compiler. This required the systems to be free from compilation errors.
2. **Bytecode Analysis:** This analysis reviewed the Java bytecode resulting from compilation and weaving to locate aspects, advices, classes, and methods in the system and store them in a recognizable format. This phase also determined interactions that existed in the bytecode among the advices and methods and among classes and aspects.
3. **Vector Space Model Creation:** Based on the list of aspects, advices, classes,

and methods, and the interactions among them, this step derived vector space models to serve as inputs to the clustering analysis.

4. **Clustering Analysis and Evaluation:** The process employed the *k*-means++ partitional clustering algorithm using the vector space models from the previous step. The selected value for K came from selecting the K with the lowest SD index value from a range of plausible K values. Cluster evaluation compared clustering results using the RS, D, DB, and SD indexes to determine validity and noted insights into potential interference.
5. **Visualization:** The resulting clusters became the input for a zoomable visualization technique designed to highlight the areas of the program most likely to exhibit interference.
6. **Assessment:** Applying this technique to two existing aspect-oriented software systems allowed for assessment of this approach in terms of both feasibility and scalability.

Model Definition

Object Models

Consider an existing aspect-oriented software system S. Because aspect interference could occur in abstractions below the aspect and class levels (Lauret et al., 2011; Tian, Cooper, Zhang, et al., 2010), the analysis considered all class methods and aspect advices to be entities in the primary object model. Class methods were defined as $CM = \{m_1, m_2, \ldots, m_p\}$, where m_i is the *i*-th of p methods and $CM \subset S$ (Şerban & Moldovan, 2006). Aspect advices were also required objects in this analysis, and were defined by $AA = \{a_1, a_2, \ldots, a_q\}$ where a_j is the j-th of q aspect advices and $AA \subset S$. The

system contained r aspects, defined by $A = \{A_1, A_2, \ldots, A_r\}$, and t classes, defined by $C = \{C_1, C_2, \ldots, C_t\}$. The primary object model for clustering was defined as $O = CM \cup AA$, containing n elements, where $n = p + q$. The secondary object model was defined as $O' = C \cup A$, containing n' elements, where $n' = t + r$.

Vector Space Models

A vector space model is a detailed way to find similarities between objects in a clustering model (Moldovan & Şerban, 2006; Şerban & Moldovan, 2006; Tribbey & Mitropoulos, 2012). Existing aspect coupling metrics, such as CAE and CDA (Ceccato & Tonella, 2004), focused on the aspect-class coupling rather than advice-method coupling (Piveta et al., 2012). Previous vector space models used in aspect mining relied upon the object-oriented fan-in-value (FIV) and fan-out-value (FOV) metrics for each object in the clustering model (Moldovan & Şerban, 2006; Şerban & Moldovan, 2006; Tribbey & Mitropoulos, 2012). While FIV and FOV provided method-method granularity, they did not account for advice-method and advice-advice interactions. Therefore, metrics IP and ICP were developed to address metric-metric, advice-metric, and advice-advice interactions and to account for both interaction tangling and interaction scattering.

Tribbey and Mitropoulos (2012) extended one-dimensional vector space models by defining a pattern matrix. This dissertation extended this idea by defining two pattern matrices, the first using IP rather than FOV, and the second based on CAE.

Given the object model $O = \{o_1, o_2, \ldots, o_n\}$, consider the $n \times n$ pattern matrix

$$P_{IP} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

where each $b_{xy} = 1, 1 \leq x \leq n, 1 \leq y \leq n$ if and only if o_x invoked object o_y, or 0

otherwise. Therefore, the sum of elements in a row r was equivalent to the IP for object o_r, and the sum of the elements in a column c was equivalent to the ICP for each object o_c. Each row (or column) in P_{IP} was considered an *n*-dimensional vector that was used as input into the clustering algorithm. For systems with high numbers of methods and advices, reducing the number of dimensions via principle component analysis (PCA) (Tribbey & Mitropoulos, 2012) was investigated.

To show how the newly defined metrics related to an existing coarsely grained metric, this research defined a second pattern matrix based on coupling on advice execution. The CAE metric counted the number of aspects that affect a module by advice, declaration constructions, and inter-type declarations (Piveta et al., 2012). Each aspect was counted only once. The pattern matrix for CAE was an $n' \times r$ matrix, defined as

$$P_{CAE} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1r} \\ b_{21} & b_{22} & \cdots & b_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n'1} & b_{p2} & \cdots & b_{n'r} \end{bmatrix},$$

where each $b_{xy} = 1, 1 \leq x \leq n', 1 \leq y \leq r$ if an only if an aspect or class O'_x ($O'_x \in C \cup A$) interacted with aspect A_y $(A_y \in A)$. Each row x in P_{CAE} represented the CAE value for aspect or class O'_x.

This research defined three vector space models—two based on pattern matrix P_{IP} and one based on pattern matrix P_{CAE}. These vector space models were:

- **Interference Potential (IP) Vector Model**: $M_{IP} = P_{IP}$.
- **Interference Causality Potential (ICP) Vector Model**: $M_{ICP} = (P_{IP})^T$.
- **Coupling on Advice Execution (CAE) Vector Model**: $M_{CAE} = P_{CAE}$.

Approach

Source Code Compilation

The first step involved compiling the source code of system S into standard Java bytecode. Variations within the source code itself, such as whitespace, commenting, and syntax (depending upon the Java version), could hinder the analysis process. However, once compiled into Java bytecode, code variations became nonexistent. The AspectJ compiler normalized aspects and advices into standard Java bytecode in a two stages (Hilsdale & Hugunin, 2004). Hilsdale and Hugunin (2004) noted that the first stage took Java and AspectJ source code and converted it directly to Java bytecode, adding annotations for non-standard Java elements like point cuts or advices. The second stage produced woven class files by inserting calls to the previously compiled advice code in the appropriate locations. The result was static bytecode that, when executed, behaved in accordance with the program's static source code. Therefore, the compiled and woven bytecode could reveal the precise method-method, advice-method, and advice-advice interactions required to complete the clustering analysis.

Bytecode Parsing

Pattern matrices P_{IP} and P_{CAE} were created directly from bytecode in two phases: the object identification phase and the interaction identification phase. Although locating methods and advices was straightforward, matrix creation involved tracking method-method, method-advice, and advice-advice coupling. Examining the compiled bytecode for method invocations provided a means of locating all items required to create the pattern matrices. Maintaining lists of interactions among advices and methods allowed for simple mappings to pattern matrices P_{IP} and P_{CAE}.

Prerequisites: All .java and .aj source files in directory *d* are compiled into .class files.
Input: File path to directory *d*
Output: In-memory listing of all classes, class methods, aspects, and advices

```
1: for each subdirectory s in directory d,
2:         for each class file c in subdirectory s,
3:                 Decompile class file c into human readable form hrf using Ruby.
4:                 Store decompilation hrf in object list OL.
5: for each object o in object list OL,
6:         if o is a class, then store in list C,
7:                 for each method m in object o, store m in list CM.
8:         else if o is an aspect, then store in list A.
9:                 for each advice name a in object o, store a in list AA.
```

Figure 1. Object Identification Phase Algorithm.

The object identification phase examined the bytecode to locate all classes, methods, advices, and aspects to create the identified vector space models. The object identification phase examined bytecode produced by the AspectJ compiler to gather the list of objects within the program under analysis. Implementation of this phase used the Ruby programming language because of its flexibility in string manipulation and the reduced overhead when compared to the Java language. The javaclass-rb Ruby gem (Kofler, 2011) provided a basis for static bytecode analysis. Because javaclass-rb did not track method invocations, the analysis phase required additional programming.

Figure 1 shows the basic steps involved in the object identification phase. The object identification phase accepted a directory path as its input. This directory and its subdirectories contained bytecode resulting from a successful compilation from source code. The algorithm walked the directory tree to the leaves and examined each class file by first converting the bytecode into an object-oriented representation. Next, the process located each class, aspect, method, and advice name within the object-oriented representation of the bytecode and stored each object into its corresponding list. Note that each method and advice was associated with its containing class or aspect. These lists

resided in memory, and allowed for interaction determination in the code.

After identifying objects contained in system S, the code analysis determined interactions among the identified objects to create pattern matrices used for vector space models. Hilsdale and Hugunin (2004) noted that every pointcut defined in the source code mapped to a corresponding static join point shadow in bytecode. The object interaction phase parsed each method in the bytecode for all invoke statements: invokedynamic, invokeinterface, invokespecial, invokestatic, and invokevirtual. Bytecode invocations accounted for method calls and executions, constructor calls and executions, advice executions, and initialization procedures (Hilsdale & Hugunin, 2004, p. 28). Bytecode used other constructs for getting and setting field values, throwing and handling exceptions, and synchronization (Hilsdale & Hugunin, 2004, p. 28), which were beyond the scope of this study.

Figure 2 shows an overview of the interaction identification phase algorithm, and may be considered an extension of the algorithm in Figure 1. The algorithm reviewed the method and advice bodies, and enumerated a list of all invocations. Each object

Prerequisites: All .java and .aj source files in directory d are compiled into .class files.
Input: Lists *OL*, C, A, CM, and AA from the object identification phase.
Output: File i, listing array m, and file i', listing array m'.

1: **declare** array m of size $|AA \cup CM|$.
2: **declare** array m' of size $|C \cup A|$.
3: **for each** class method and advice o in system S from lists CM and AA,
4: **declare** array oa, a bitmap of size $|AA \cup CM|$
5: **when** o invokes object o', where $o' \in AA \cup CM$, **then**
6: Set $oa_{o'i} = 1$, where $o'i$ is the index corresponding to object o'.
7: Add oa to m.
8: **for each** class and aspect o' in system S from lists C and A,
9: **declare** array oa', a bitmap of size $|A|$
10: **when** o' invokes object o_a, where $o_a \in AA$ and AA is an advice of aspect A_x, **then**
11: Set $oa'_{A_x i} = 1$, where $A_x i$ is the index corresponding to Aspect A_x.
12: Add oa' to m'.
13: Output list m to file i and m' to file i'.

Figure 2. Interaction Identification Phase Algorithm.

interaction was recorded into a one-dimensional bitmap oa, of size $|AA \cup CM|$ for each object o_x. Placing a 1 at location oa_y indicated the invocation of object o_y by object o_x. Array oa was stored at location m_x and the process continued for all objects. One-dimensional array oa' of size $|A|$ was defined as a bitmap of class-aspect interactions for each class or aspect o'_x. Placing a 1 at location oa'_y indicated an interaction between class or aspect o'_x and aspect A_y. Array oa' was stored at location m'_x and the process continued for all objects. At the end of the procedure, two-dimensional array m represented pattern matrix P_{IP} by denoting object invocations, where location m_{xy} contained 1 if o_x invoked o_y and 0 otherwise. Two-dimensional array m' represented pattern matrix P_{CAE}, where location m'_{xy} contained 1 if class or aspect o'_x was affected by aspect A_y or 0 otherwise. The process saved both matrices to files for use by the clustering algorithm.

Vector Space Model Creation

Vector space models M_{IP}, M_{ICP}, and M_{CAE} were defined from two-dimensional arrays m and m' in the previous step. Model M_{IP} was a direct mapping of array m while model M_{ICP} was a direct mapping of the transposition of array m. Models M_{IP} and M_{ICP} contained n data points with n dimensions. Model M_{CAE} was a direct mapping of array m', and contained n' data points with r dimensions.

Principal Component Analysis

Because of the potential for a large number of dimensions in both M_{IP} and M_{ICP}, this research used principal component analysis (PCA) in an attempt to reduce the number of dimensions. PCA computed eigenvalues for the covariance of each pattern matrix. These eigenvalues also represented the estimated variances of the converted

variables. The sum of all eigenvalues constituted the overall variation within the matrix. Many components contributed to less than 1% of the total variation, so the process retained only components contributing a variance greater than 1%. Clustering analysis was then applied to the PCA-reduced matrices.

Clustering Analysis

The *k*-means algorithm, a partitional clustering method, was employed to locate potential aspect interference. When using *k*-means, determining the initial number of clusters K to pass into the algorithm was of concern. Şerban and Moldovan (2006) selected initial centers by maximizing the minimum distances between the centroid and the clustering objects until reaching a minimum threshold. Tribbey and Mitropoulos (2012) used the *k-means*++ algorithm (Arthur & Vassilvitskii, 2007) to seed initial centers, and the SD index (Maria Halkidi, Batistakis, & Vazirgiannis, 2002; M. Halkidi, Vazirgiannis, & Batistakis, 2000) (which incorporated cluster density and variance) to determine an optimal number of clusters.

This study used the SD index and the *k*-means++ seeding approach to determine the number of clusters K that was fed into the *k*-means++ algorithm. The clustering steps, adapted from Tribbey and Mitropoulos (2012), included the following.

1. To determine the minimum (K_{min}) and maximum (K_{max}) values for K, let K_μ be the number of unique values for the coupling metric (IP, ICP, or CAE) that exist in the vector space model. Set K_{min} and K_{max} as follows:
    a. $K_{min} = \max\left(2, \left(K_\mu - 20\right)\right)$
    b. $K_{max} = (K_\mu + 10)$

Note that the *k*-means++ algorithm required $2 \leq K_{min} \leq K_{max} \leq N$, where N

was the number of vectors in the vector space model.

2. For each value of K between K_{min} and K_{max}, execute the *k*-means++ algorithm five times and record the minimum SD index value. The value of K that produced the lowest SD index between K_{min} and K_{max} became the chosen K, denoted K^*.
3. Perform the *k*-means++ algorithm 100 times using K^*. Record clustering metrics (described in the next section) and clusterings for each individual run.
4. Compute the mean, median, and standard deviation of the values collected for each clustering metric. Use Wilcoxson rank sum testing to compare resulting clustering metrics.

The SD index, proposed by Halkidi, Vazirgiannis, and Batistakis (2000), measured the average scattering for a cluster as well as the total separation between clusters. Average scattering for a clustering was defined as

$$Scat(n_c) = \frac{1}{n_c}\sum_{i=1}^{n_c}\frac{\|\sigma(v_i)\|}{\|\sigma(\mathrm{X})\|}$$

where $\sigma(v_i)$ was the variance within cluster i and $\sigma(\mathrm{X})$ was the total variance in the clustering. Total separation between clusters was defined as

$$Dis(n_c) = \frac{D_{max}}{D_{min}}\sum_{k=1}^{n_c}\left(\sum_{z=1}^{n_c}\|v_k - v_z\|\right)^{-1}$$

where D_{max} was the maximum distance between clusters, $D_{max} = \max(\|v_i - v_j\|)\ \forall i,j \in \{1,2,3,\dots,n_c\}$, and D_{min} was the minimum distance between clusters, $D_{min} = \min(\|v_i - v_j\|)\ \forall i,j \in \{1,2,3,\dots,n_c\}$. The SD index was defined as

$$SD(n_c) = \alpha \times Scat(n_c) + Dis(n_c)$$

where $\alpha = Dis(\max(n_c))$. (Maria Halkidi et al., 2002; M. Halkidi et al., 2000)

The *k*-means++ algorithm, introduced by Arthur and Vassilvitskii (2007), combined a probability-based seeding technique to determine initial centroids for the *k*-means algorithm. While *k*-means chose K random centroids and adjusted them repeatedly until they remain unchanged, *k*-means++ chose one random centroid and each subsequent centroid using a probability formula known as D^2 *weighting* (Arthur & Vassilvitskii, 2007). This technique allowed the *k*-means algorithm to stop sooner (on average) than with randomly seeded centroids, resulting in increased speed and accuracy.

The R Project for Statistical Computing ("The R Project for Statistical Computing," 2013) included function SDIndex in library BCA, which plotted the value of the SD index for different values of K over a common data set. R package flexclust included function kcca, which could perform *k*-means clustering with *k*-means++ seeding. The value of K^* for each vector space model came from executing the kcca clustering function repeatedly between K_{min} and K_{max} and finding the lowest value for SD using the SDIndex functionality.

The kcca function (using the *k*-means algorithm with *k*-means++ seeding) was run 100 times for each vector space model, passing K^* as the number of centers. For each run, the program collected clusters and the various clustering metrics for use in validation and analysis.

Validation and Empirical Analysis

Halkidi, Batistakis, and Vazirgiannis (2001) conducted surveys of cluster validation techniques, noting most assessments investigated both compactness within a cluster, and the separation between clusters. The best clustering results revealed compact clusters with high separation. To assess clustering validity, Tribbey and Mitropoulos

(2012) used the *R-squared* (RS) index. The RS index is the ratio of the sum of square errors between clusters to the total sum of square errors (Maria Halkidi et al., 2001). The RS index was formally defined as

$$RS = \frac{SS_b}{SS_t} = \frac{SS_t - SS_w}{SS_t}$$

where SS_b was the sum of square errors between clusters, and SS_t was the total sum of square errors. SS_t was defined as follows.

$$SS_t = SS_b + SS_w = \sum_{j=1}^{v} \left[\sum_{i=1}^{n_c} (X_i - \bar{X})^2 \right]$$

$$SS_w = \sum_{\substack{i=1\ldots c \\ j=1\ldots v}} \left[\sum_{k=1}^{n_{ij}} \left(X_k - \hat{X}_k \right)^2 \right]$$

Clusterings with higher RS index values were more uniform because they indicated a larger distance between groups (Maria Halkidi et al., 2001). Clusterings with lower RS values would indicate less compactness and may indicate higher interference potential across the system, while higher RS values would indicate more compactness and more localized interference potential among objects within clusters.

This study also assessed the validity of the clustering results by using two classic measurements—*Dunn's Index* (D) and *Davies-Bouldin's Index* (DB) (Maria Halkidi et al., 2001). Dunn's Index computed a value based on the ratio of cluster dissimilarity to the maximum cluster diameter, and was defined as

$$D_{n_c} = \min_{i=1\ldots n_c} \left(\min_{j=i+1,\ldots,n_c} \frac{\left(d(c_i, c_j) \right)}{\max\limits_{k=1,\ldots,n_c} \operatorname{diam}(c_k)} \right)$$

Large values for D indicated compact clusters with high separation. Davies-Bouldin's

Index computed a value based on the average similarities between each cluster and its most-similar cluster, and was defined as

$$DB_{n_c} = \frac{1}{n_c}\sum_{i=1}^{n_c} \max_{i=1,\ldots,n_c; i\neq j}\left(\frac{(s_i + s_j)}{d_{ij}}\right)$$

where s_i and s_j are the dispersion of clusters i and j, and d_{ij} is the dissimilarity between clusters i and j. The best clustering results would have low values for DB, indicating the average cluster was compact with high distances between groups. (Maria Halkidi et al., 2001)

Together, RS, D, DB, and the previously described SD index provided a solid empirical evaluation designed to determine which clustering vector model produced the best results and what the results indicated about overall aspect interference within the system S. Low values for RS and D showed similar clusters with potential interference spread across multiple clusters, while high RS and D values indicated interference potential localized within clusters. Low values for DB and SD showed more localized interference potential, while higher values for DB and SD indicated scattered interference potential across the system. Table 4 gives further implications of each metric used to interpret clustering results.

Given these metrics used, one-tailed Wilcoxson rank sum testing provided a way to compare two sets of metrics based on expected outcomes. Given two sets of statistics, A and B, the one-tailed Wilcoxson rank sum test ranks all values and determines the probability that the ranking occurred by chance. Thus, low *p*-values indicated a low probability for a random result and suggested that the shift in rank is statistically significant. The hypotheses tested for this study were:

- The PCA-reduced model produced a better clustering than the non-reduced model.
- The modified program model produced a better clustering than the unmodified program model.

Table 4. A Summary of Clustering Metrics Used in this Study.

Metric	Description	Range	Best Value	Implications
$\boldsymbol{RS}$	Cluster dissimilarity based on sum of squares	0-1	High	High values indicate interference localized within a small number of clusters.
$\boldsymbol{D}$	Cluster distances compared to cluster diameters	≥ 0	High	High values indicate small clusters that are well-separated. More uniform clusters would indicate that interference is spread across the system.
$\boldsymbol{DB}$	Average similarity between each cluster and it's next most similar cluster	≥ 0	Low	Lower values show that clusters are less similar, indicating that interference would be localized to a small number of clusters.
$\boldsymbol{SD}$	Sum of average cluster scattering and total cluster separation.	≥ 0	Low	Average cluster scattering is the ratio of the cluster variance to the total variance. Therefore, low values for SD would indicate more uniform clusters with localized interference problems. As SD increases, interference problems increase across clusters.

Table 5 displays a sample of how rank sum test results appear. Note that each column contains the *p*-value result of the one-tailed test. Both RS and D expect set B to contain higher values, while DB and SD expect set B to contain lower values, as indicated in the top row. Results less than 0.01 were significant enough to accept the alternative hypothesis tested. While *p*-values cannot be zero, many of the results shown appear as zero because they were so small.

Table 5. Wilcoxon Rank Sum Test Results (Sample)

	A < B		**A > B**	
	RS	**D**	**DB**	**SD**
AT α M_{IP}, AT α $M_{IP}\Pi$				
AT β M_{IP}, AT β $M_{IP}\Pi$				
AT α M_{IP}, AT β M_{IP}				
AT α M_{IP} Π, AT β M_{IP} Π				

Visualization

Clusterings that represented the best choice (as defined by the highest RS metric)

fed into the visualization component so programmers could observe the clusters in a graphical way. The purpose of this visualization was to assist in the assessment of the results, and to increase understanding of the results. The visualization utilized the D3.js JavaScript Library (M Bostock, 2012). This library contained visualization models that allow programmers to zoom into clusters to reveal more detail. This extended the idea of Fabry et al. (2011), who validated a zooming visualization technique based on mapping applications.

Assessment and Data Analysis

The results of the study included an assessment of both the approach's feasibility and its scalability. To determine feasibility, the study required a smaller-scale AspectJ program. The study also required a larger AspectJ program to ensure scalability. Several open-source AOP systems existed that were potentially suitable for this study. Apel (2010) provided an analysis of eleven AspectJ programs: AspectTetris, OAS, Prevayler, AODP, FACET, ActiveAspect, HealthWatcher, AJHotDraw, Hypercast, AJHSQLDB, and Abacus. Of these, only five were not AOP refactorings of existing OOP systems: AspectTetris, OAS, FACET, ActiveAspect, and HealthWatcher. Each of these programs had an approximate 80% class code to 20% aspect code ratio, except HealthWatcher. Despite being a refactored project, AJHotDraw included over 22,000 lines of code, and has had wide use throughout the AOP literature. Therefore, AJHotDraw was chosen for large-scale testing. A smaller code base allowed easier feasibility testing, so this study used AspectTetris to assess feasibility. (Apel, 2010)

To assist with the analysis, both programs were cloned, and a new aspect was added to each clone. The new aspect contained one before advice that resulted in an

interaction between many methods and the advice, leading to a high ICP value. Each aspect also contained an after advice that made a method call with the hope of ensuring a high IP value.

An example presentation of dimensions and indicated values of *K* appears in Table 6. The *k*-means++ algorithm was run 100 times to produce the results for the indicated *K*. Each vector space model produced specific RS, D, DB, and SD values, which were used for comparison and validation. The resulting mean, median, and standard deviation are presented for each clustering metric, as shown in Table 7.

Table 6. Dimensions and indicated K for Program 1 (Sample).

Model	Dim	K*
$\boldsymbol{M_{IP}}$		
$\boldsymbol{M_{ICP}}$		
$\boldsymbol{M_{CAE}}$		

Table 7. Summary Statistics after 100 Runs of K-means++ (Sample).

Measure	Model	Mean	Median	Std. Dev.
$\boldsymbol{RS}$	M_{IP}			
	M_{ICP}			
	M_{CAE}			
$\boldsymbol{D}$	M_{IP}			
	M_{ICP}			
	M_{CAE}			
$\boldsymbol{DB}$	M_{IP}			
	M_{ICP}			
	M_{CAE}			
$\boldsymbol{SD}$	M_{IP}			
	M_{ICP}			
	M_{CAE}			

Resource Requirements

With a few exceptions such as Figueroa (2013) and Figueroa, Tabareau, and Tanter (2013), studies involving AOP used the AspectJ language to conduct experiments (Lauret et al., 2011). Therefore, this study utilized AspectJ (v. 1.8.x) as the platform for experimental purposes, though the general findings are applicable to other AOP languages. The AspectJ compiler utilized Java SE 6 on Mac OS X for source code compilation into bytecode. The Ruby programming language (v. 2.0.x), and portions of

the javaclass-rb gem (Kofler, 2011) were used for bytecode analysis. The R Project for Statistical Computing (2013) provided clustering and clustering metric resources. The D3.js JavaScript library (M Bostock, 2012) provided the means for visualization. Therefore, no special resources other than a standard computer system with Java and AspectJ compilers and a JavaScript-enabled web browser were necessary for completing this work.

Summary

The overall approach for this study required AOP source code compilation into Java bytecode using the AspectJ compiler. Using static bytecode analysis, the program created the described vector space models that fed into the *k*-means++ clustering algorithm. To determine an optimal value for *K*, clustering was run 5 times for each potential value of *K* and the *K* having the lowest SD index value was retained. Clustering validity metrics assessed the clustering model with the best performance, and illustrated how well clustering detected the introduction of aspect interference in a program. Visualizations of clusterings with the best fit based on the RS metric assisted in showing the potential problem areas within the programs.

Chapter 4

Results

This chapter presents the results of this study. Following a description of the notation used in the results, general observations are made about each benchmark application. Then, results for each application are presented and discussed. Finally, a summary of the results concludes this chapter.

Vector Space Model Notation

Three vector space models described the data for both AspectTetris (AT) and AJHotDraw (AJHD). Because each program included an unmodified (α) and a modified (β) version, vector space models in the results used the following notation.

- $\alpha\ M_{IP}$ - The interference potential matrix for the unmodified program.
- $\alpha\ M_{ICP}$ - The interference causality potential matrix for the unmodified program.
- $\alpha\ M_{CAE}$ - The coupling on advice execution matrix for the unmodified program.
- $\beta\ M_{IP}$ - The interference potential matrix for the modified program.
- $\beta\ M_{ICP}$ - The interference causality potential matrix for the modified program.
- $\beta\ M_{CAE}$ - The coupling on advice execution matrix for the modified program.

Adding Π to the notations denoted matrices reduced by PCA.

Data Presentation

When possible, data were presented in tabular format. However, in a few cases, visual representation was necessary. Plots appearing in this chapter used dotted lines to

connect data points representing SD index values. These dotted lines appear only to assist in visualizing the results, and imply no other relationships between these data points. Visualizations of resulting clusters used the D3 JavaScript library (M Bostock, 2012). Zoomable Circle Packing, an open-source example created by Mike Bostock (2013), produced a zoomable hierarchical visualization that read json code. Json code was rendered in the R statistical software ("The R Project for Statistical Computing," 2013) and fed into a local version of the visualization program. While these visualizations were interactive via the web browser, the images presented here show the fully zoomed-out versions. Because of this non-interactive medium, some visualizations may appear difficult to read, but are discussed in the text.

Application Characteristics

Two applications were used to complete this study. The first was AspectTetris (AT) (Evertsson, 2003), an AspectJ implementation of the game Tetris. Following compilation, the bytecode of the unmodified AT application contained 158 objects among 8 aspects and 16 classes. To assess the research questions posed, a new aspect, SeedAspect, was added to the AT code that contained one generic point cut for every method call, and two advices. A before advice simply output a string to the console for the point cut, while an after advice called a method within the SeedAspect code for the given point cut. The before advice was designed to have an increased ICP value, while the method call from the after advice was expected to raise the advice's IP value. Following compilation, the modified version of AT contained 166 objects among 9 aspects and 16 classes.

The second application was AJHotDraw v.0.4 (AJHD) ("AJHotDraw," 2007), an

AspectJ program based on JHotDraw for drawing images. Following compilation, the bytecode of the unmodified version of AJHD contained 3,953 objects among 31 aspects and 396 classes. The same aspect—SeedAspect (described above)—was added to the AJHD code. After compilation, the modified version of the program contained 4,037 objects among 32 aspects and 407 classes. Note that the increase in the number of classes was due to aspect weaving at compile time. The AspectJ compiler broke classes into subsets as needed to accommodate advice code.

AspectTetris Results

AT with Model M_{IP} (Interference Potential)

Interference Potential (IP) counted the number of advices and methods the given advice or method invoked. Following compilation, program AT α consisted of 158 methods and advices, resulting in the 158×158 model AT α M_{IP}. Program AT α had 10 unique values for the IP metric. Compilation of program AT β produced 166 methods and advices, resulting in the 166×166 model AT β M_{IP}. Program AT β included 13 unique values for the IP metric. Based on these values, Table 8 shows the range of *K* values tested during the SD analysis to determine K^*.

Table 8. Values of K Tested for AspectTetris Model M_{IP}

Model	K_{μ}	K_{min}	K_{max}
AT α M_{IP}	10	2	20
AT β M_{IP}	13	2	23

PCA of the unmodified and modified versions of the AspectTetris program reduced dimensions for M_{IP} in both cases. Table 9 shows the dimension reduction achieved and the values found for K^*. Figure 3 plots the SD validity index values for the unmodified version of the program, while Figure 4 displays the SD validity index values for the modified version of the program. SD analysis indicated a value of $K^* = 4$ for

Table 9. Suggested Numbers of Clusters for AspectTetris Model M_{IP}

Model	Dim	K^*
AT α M_{IP}	158	4
AT α $M_{IP}\Pi$	19	5
AT β M_{IP}	166	5
AT β $M_{IP}\Pi$	20	7

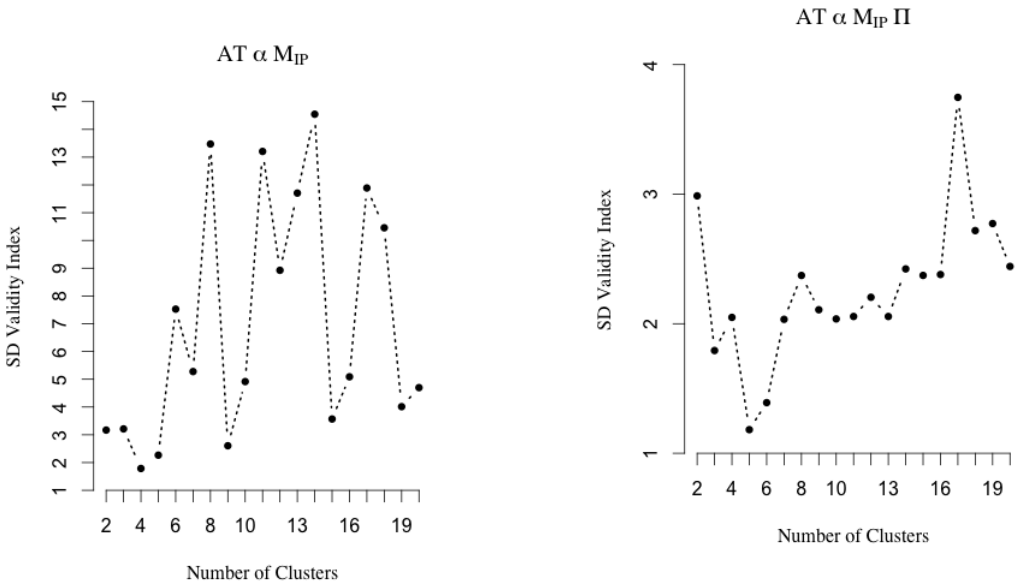


Figure 3. SD validity index values for the AT α M_{IP} and the AT α M_{IP} Π models.

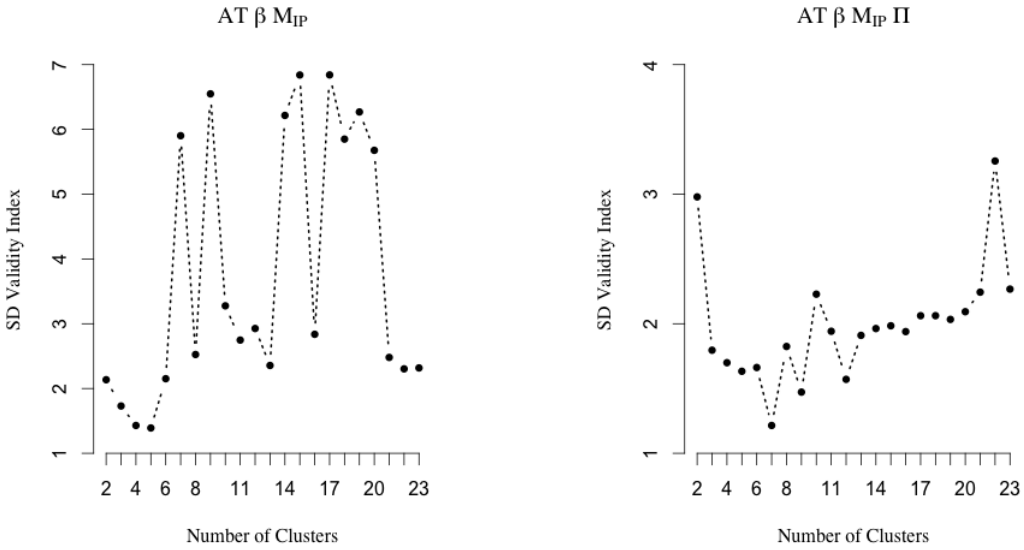


Figure 4. SD validity index values for the AT β M_{IP} and the AT β M_{IP} Π models.

model $AT\ \alpha\ M_{IP}$, $K^* = 5$ for model $AT\ \alpha\ M_{IP}\Pi$, $K^* = 5$ for model $AT\ \beta\ M_{IP}$, and $K^* = 7$ for model $AT\ \beta\ M_{IP}\ \Pi$.

Using the suggested K values, summary statistics were collected for each IP-based model and are recorded in Table 10. Resulting *p*-values from the one-tailed Wilcoxon rank sum tests appear in Table 11. Except for Dunn's index, metric mean scores indicated that program AT β produced more favorable clusterings than program AT α, and suggested that PCA-reduced model AT $\beta\ M_{IP}\Pi$ had the best overall clustering. This was confirmed by significant *p*-values for the RS and SD metrics. Regarding cluster similarity, the RS metric showed that clusters in model AT $\beta\ M_{IP}$ were more dissimilar than model AT $\alpha\ M_{IP}$, which agreed with the DB and SD metrics. This indicated that the modifications to program AT β had the intended effect of introducing potential interference, and that clustering successfully detected the interference. Metric D indicated a negligible improvement for model AT $\beta\ M_{IP}$ over AT $\alpha\ M_{IP}$ in mean scores, and showed no significant shift. Because of the Dunn index's sensitivity to outliers (Maria Halkidi et al., 2002), Dunn index results for models AT $\alpha\ M_{IP}\Pi$ and AT $\beta\ M_{IP}\Pi$ were thought to provide more accurate representations since PCA reduction diminishes noise within the data. However, comparison of the PCA-reduced models showed no significant improvement for the Dunn index for program AT α over program AT β. This agreed with Maria Halkidi et al (2002), who suggested that Dunn's index was sensitive to the chosen value of K because of its dependence on cluster diameter.

Clusterings with the highest RS values were chosen for visualization. Figure 5 shows the best clusterings for program AT α. Note that in all cases, method

Table 10. Summary Statistics for AT Model M_{IP} after 100 Runs of K-means++

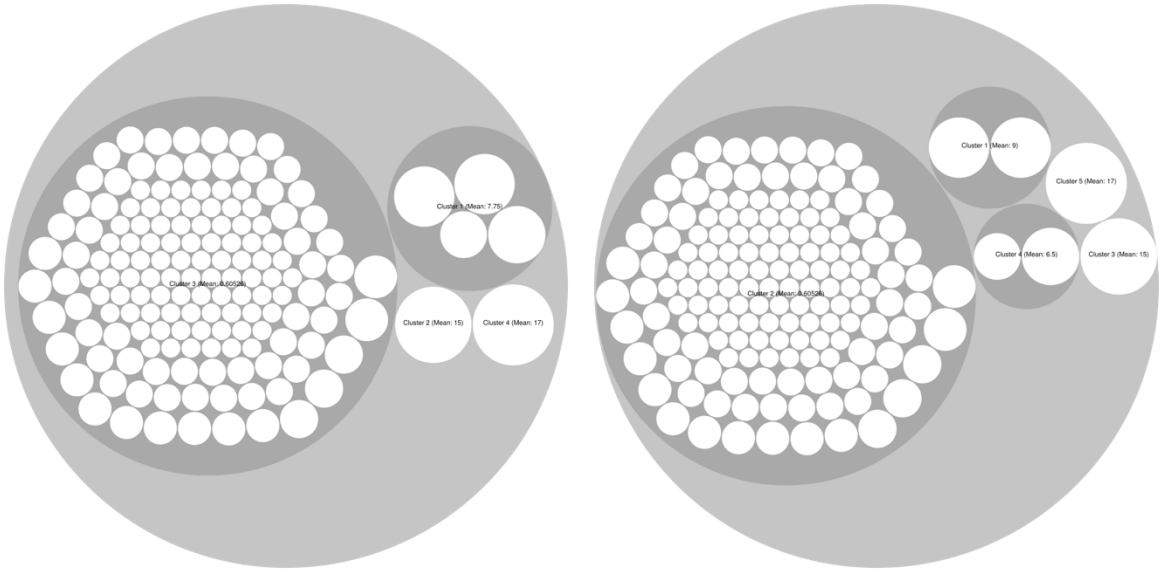
Measure	Model	Mean	Median	Std. Dev.
RS	AT α M_{IP}	0.2586	0.2609	0.0241
	AT α $M_{IP}\Pi$	0.4414	0.4512	0.0171
	AT β M_{IP}	0.5590	0.5553	0.0180
	AT β $M_{IP}\Pi$	0.7440	0.7467	0.0080
D	AT α M_{IP}	0.3658	0.3430	0.1712
	AT α $M_{IP}\Pi$	0.3207	0.2412	0.1448
	AT β M_{IP}	0.3212	0.3015	0.1123
	AT β $M_{IP}\Pi$	0.2780	0.2649	0.0601
DB	AT α M_{IP}	1.5550	1.4320	0.5039
	AT α $M_{IP}\Pi$	1.0310	1.0180	0.1366
	AT β M_{IP}	1.1980	1.1690	0.2177
	AT β $M_{IP}\Pi$	0.9871	1.0090	0.1280
SD	AT α M_{IP}	5.0244	5.0538	1.6244
	AT α $M_{IP}\Pi$	3.7879	3.9227	0.5001
	AT β M_{IP}	2.4745	2.4236	0.6723
	AT β $M_{IP}\Pi$	2.1400	2.1488	0.3774

Table 11. Wilcoxon Rank Sum Test *p*-Values for AT IP Models

	A < B		**A > B**	
	RS	**D**	**DB**	**SD**
AT α M_{IP}, AT α $M_{IP}\Pi$	0.0000	0.6791	0.0000	0.0000
AT β M_{IP}, AT β $M_{IP}\Pi$	0.0000	0.9404	0.0000	0.0024
AT α M_{IP}, AT β M_{IP}	0.0000	0.9523	0.0002	0.0000
AT α M_{IP} Π, AT β M_{IP} Π	0.0000	0.0109	0.0071	0.0000

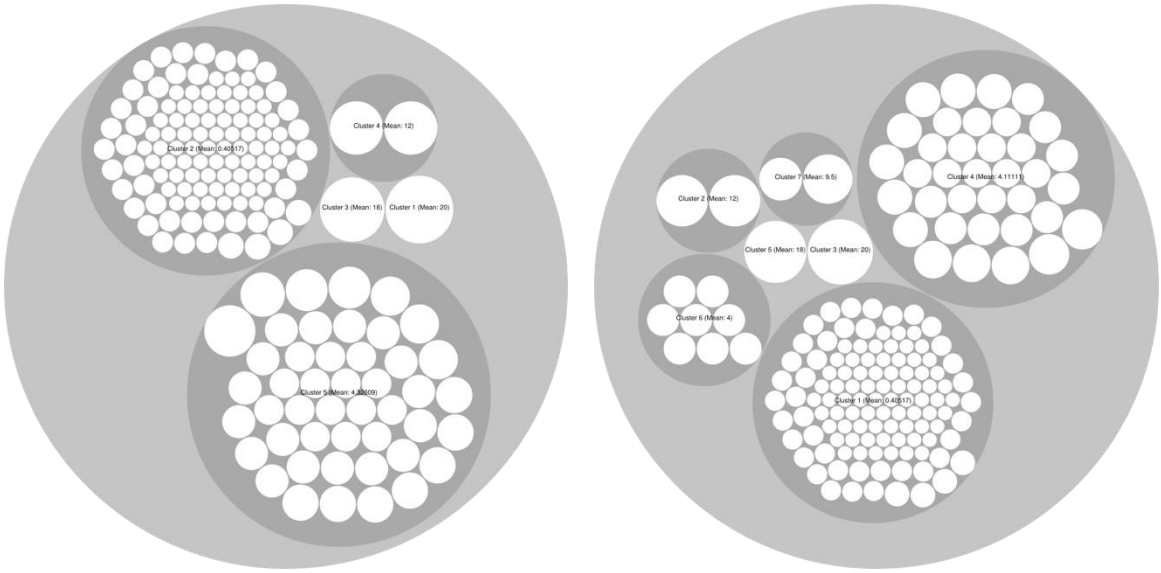
Gui/TetrisGUI.<init> was the sole member of the highest-ranked cluster, with an IP value of 17. The second-ranked cluster contained AspectTetris.incomingEvent, a method with an IP value of 15. Methods AspectTetris.newBlock and AspectTetris.startTetris (both with IP values of 9) appeared in the third-ranked cluster. Thus, the initialization of the GUI and the AspectTetris classes included the highest interference potential within the unmodified program.

Figure 6 shows suggested best clusterings for program AT β based on the IP metric. Note that the large cluster depicted in Figure 5 appears to be split into two clusters in Figure 6. The fourth-ranked cluster of model AT β M_{IP}, and the fifth-ranked cluster for model AT β M_{IP} Π had a mean IP of around 4, and included the seeded aspect advices from SeedAspect. This indicated that modifications to AspectTetris increased IP values overall, and this overall increase was detected by clustering.



Model AT α M_{IP} Model AT α $M_{IP}\Pi$

Figure 5. Visualizations of Best IP Clusterings for Program AT α.



Model AT β M_{IP} Model AT β $M_{IP}\Pi$

Figure 6. Visualizations of Best IP Clusterings for Program AT β.

AT with Model M_{ICP} (Interference Causality Potential)

Interference Causality Potential (ICP) counted the number of advices or methods that called the given advice or method. Because the ICP pattern matrix was the transposition of the IP pattern matrix, AT α M_{ICP} was 158×158, while AT β M_{ICP} was

166×166. Program AT α contained 8 unique values for ICP, while program AT β contained 9 unique values for ICP. Table 12 shows the range of *K*-values tested for K in the SD index analysis.

Table 12. Values of K Tested for AspectTetris Model M_{ICP}

Model	K_{μ}	K_{min}	K_{max}
AT α M_{ICP}	8	2	18
AT β M_{ICP}	9	2	19

Table 13 shows that PCA analysis reduced the dimensions of both the modified and unmodified program models, and summarizes the value of K for each model. Interestingly, each model indicated that a very small number of clusters were sufficient to model the data for vector space model M_{ICP}. Figure 7 plots SD validity index values for AT α, while Figure 8 plots SD validity index values for AT β.

Table 13. Suggested Numbers of Clusters for AspectTetris Model M_{ICP}

Model	Dim	K^*
AT α M_{ICP}	158	3
AT α $M_{ICP}\Pi$	19	3
AT β M_{ICP}	166	3
AT β $M_{ICP}\Pi$	20	2

Table 14 shows summary statistics for models based on interference causality potential, and Table 15 shows *p*-values for Wilcoxon rank sum tests. All metrics revealed significant improvements for program AT β over AT α for the unreduced and reduced models. D indicated a slight improvement for AT β M_{ICP} over AT α M_{ICP}, likely because the number of clusters tested was the same in both cases. For the reduced models, D was in favor of AT α $M_{ICP}\Pi$ over AT β $M_{ICP}\Pi$, which may indicate the latter had clusters of smaller diameters due to different values for K, though the *p*-value for this comparison was within the significant range. RS and DB agreed that program AT β had more dissimilar clusters than program AT α. SD results also showed more compact and

separated clusters in program AT β. This indicated strong support for clustering based on ICP, and indicated its validity as a method for locating the causes of potential interference within a system.

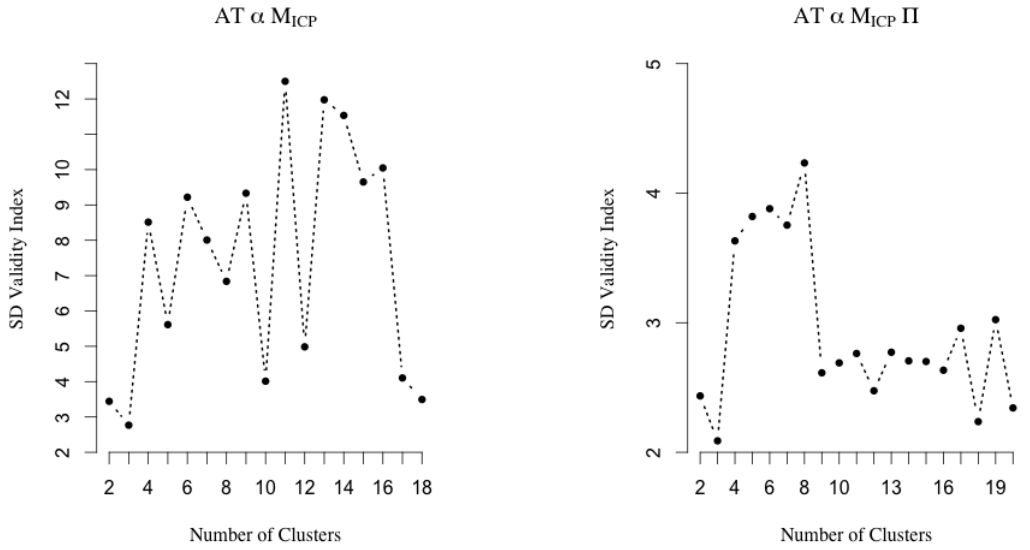


Figure 7. SD validity index values for the AT α M_{ICP} and the AT α M_{ICP} Π models.

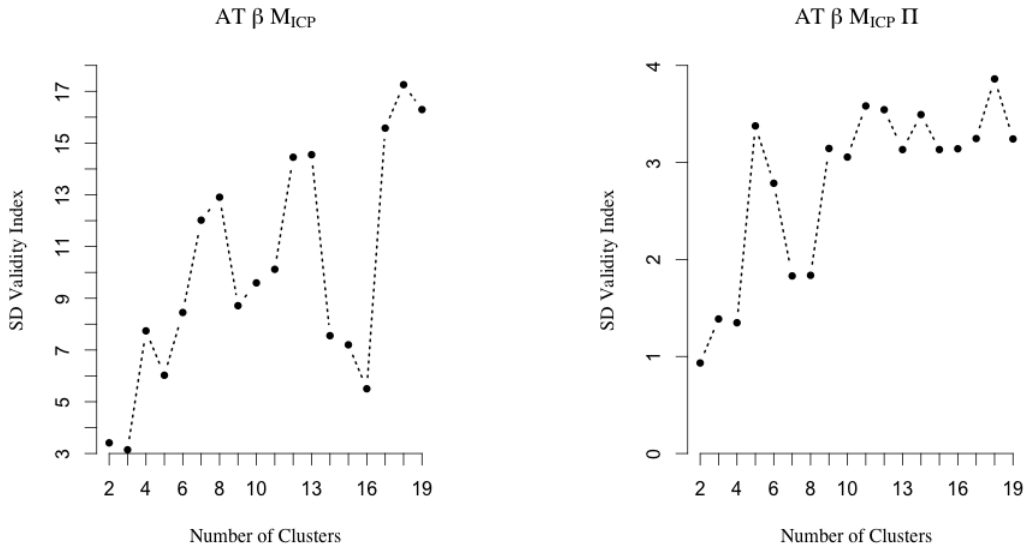


Figure 8. SD validity index values for the AT β M_{ICP} and the AT β M_{ICP} Π models.

Table 14. Summary Statistics for AT Model M_{ICP} after 100 Runs of K-means++

Measure	Model	Mean	Median	Std. Dev.
RS	AT α M_{ICP}	0.1429	0.1438	0.2459
	AT α $M_{ICP}\Pi$	0.1771	0.1703	0.0289
	AT β M_{ICP}	0.5228	0.5228	0.0109
	AT β $M_{ICP}\Pi$	0.5670	0.5670	0.0000
D	AT α M_{ICP}	0.3166	0.2887	0.0699
	AT α $M_{ICP}\Pi$	0.3076	0.2875	0.0394
	AT β M_{ICP}	0.3392	0.2887	0.0978
	AT β $M_{ICP}\Pi$	1.7387	1.7387	0.0000
DB	AT α M_{ICP}	1.8277	1.7871	0.4518
	AT α $M_{ICP}\Pi$	1.3924	1.4508	0.3037
	AT β M_{ICP}	1.4478	1.3485	0.3157
	AT β $M_{ICP}\Pi$	0.1175	0.1175	0.0000
SD	AT α M_{ICP}	3.7157	3.7951	0.7810
	AT α $M_{ICP}\Pi$	3.8371	3.0348	1.3126
	AT β M_{ICP}	3.1407	2.8705	0.5994
	AT β $M_{ICP}\Pi$	2.1401	2.1488	0.3774

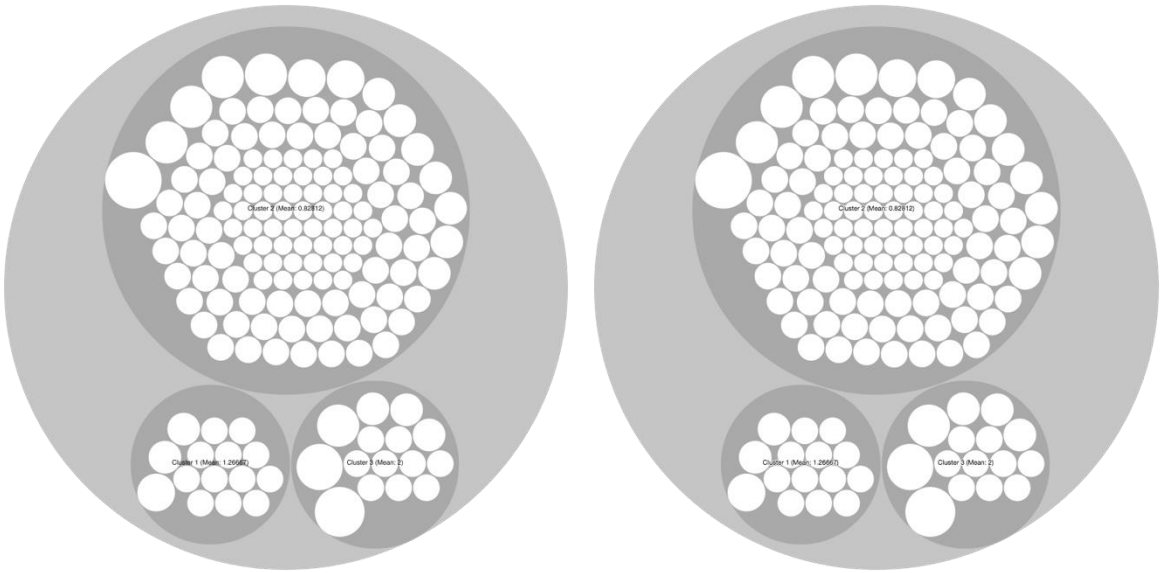
Table 15. Wilcoxson Rank Sum Test *p*-Values for AT ICP Models

	A < B		**A > B**	
	RS	**D**	**DB**	**SD**
AT α M_{ICP}, AT α $M_{ICP}\Pi$	0.0000	0.3322	0.0000	0.2187
AT β M_{ICP}, AT β $M_{ICP}\Pi$	0.0000	0.0000	0.0000	0.0000
AT α M_{ICP}, AT β M_{ICP}	0.0000	0.3548	0.0000	0.0000
AT α M_{ICP} Π, AT β M_{ICP} Π	0.0000	0.0000	0.0000	0.0000

Figure 9 shows the best ICP clusterings for the unmodified version of the program based on the clustering with the highest RS value. Both the reduced and unreduced models produced the same result. These results suggested that the unmodified version of the program included only a few methods with high ICP. The method with the highest ICP was generated by the AspectJ code as an implementation of the Singleton pattern. Aspects/Highscore/Levels.aspectOf had an ICP value of 6, indicating the Aspects/Highscore/Levels aspect had the highest potential to cause interference problems.

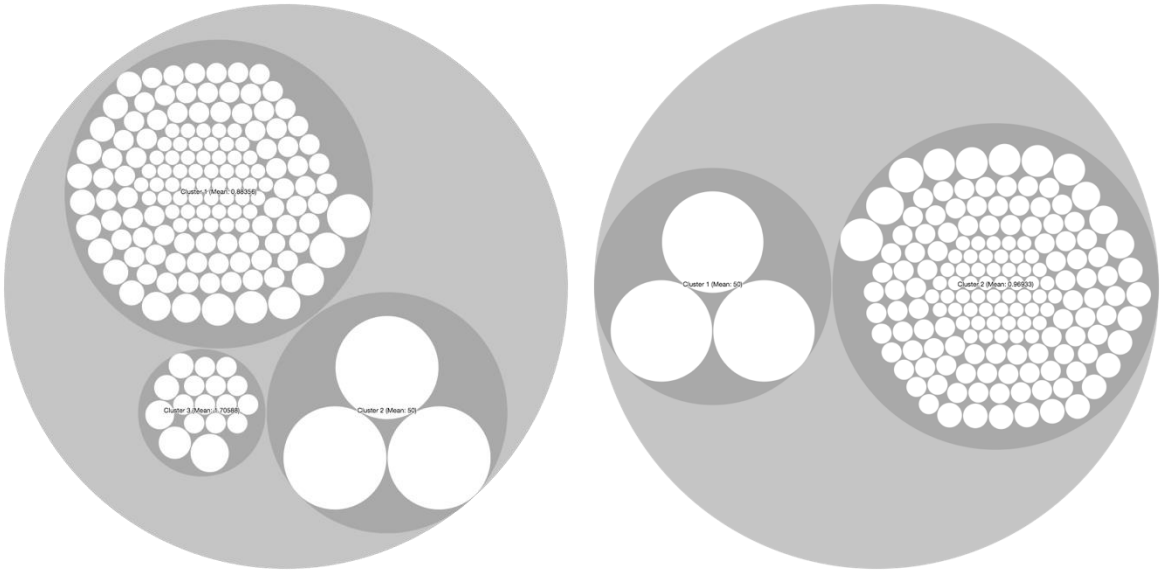
Figure 10 shows the best ICP clusterings for the modified version of AspectTetris based on the highest RS value. The highest-ranked cluster contained three bytecode methods in both AT β ICP models, each with an ICP value of 50. All three bytecode methods came

from Aspects/SeedAspect, the aspect added to increase the chance of potential interference. Therefore, the ICP clustering model correctly detected and identified the introduction of aspects with high ICP.



Model AT α M_{ICP} Model AT α M_{ICP} Π

Figure 9. Visualizations of Best ICP Clusterings for Program AT α.



Model AT β M_{ICP} Model AT $\beta M_{ICP} \Pi$

Figure 10. Visualizations of Best ICP Clusterings for Program AT β.

AT with Model M_{CAE} (Coupling on Advice Execution)

Coupling on Advice Execution (CAE) counted the number of aspects that contain

advices triggered by the advices or methods of a given aspect or class (Ceccato & Tonella, 2004). CAE was more coarsely grained than either IP or ICP, but was tested to see whether it was effective for showing potential interference problems at the class or aspect level. Vector space model M_{CAE} was 24×8 for the unmodified version of the AspectTetris program and 25×9 for the modified version. Table 16 shows that there were 4 unique values for CAE in program AT α and 5 unique values for AT β. Because these matrices were so small, K_{max} was set as the maximum possible number for K allowed by the clustering procedure, which was the number of unique vectors in the matrix.

Table 16. Values of K Tested for AspectTetris for Model M_{CAE}

Model	K_{μ}	K_{min}	K_{max}
AT α M_{CAE}	4	2	11
AT β M_{CAE}	5	2	12

Table 17. Suggested Numbers of Clusters for Aspect Tetris Model M_{CAE}

Model	Dim	K^*
AT α M_{CAE}	8	3
AT α $M_{CAE}\Pi$	-	-
AT β M_{CAE}	9	3
AT β $M_{CAE}\Pi$	-	-

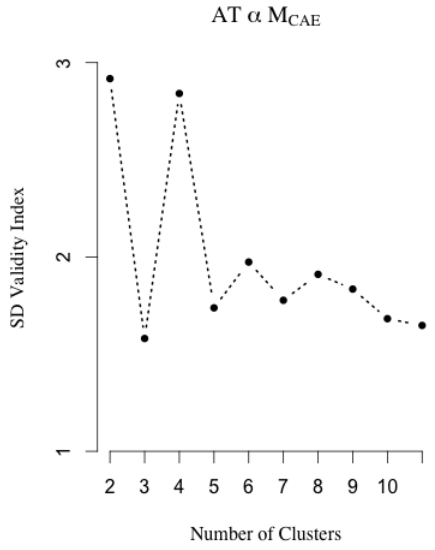


Figure 11. SD validity index values for the AT α M_{CAE} model.

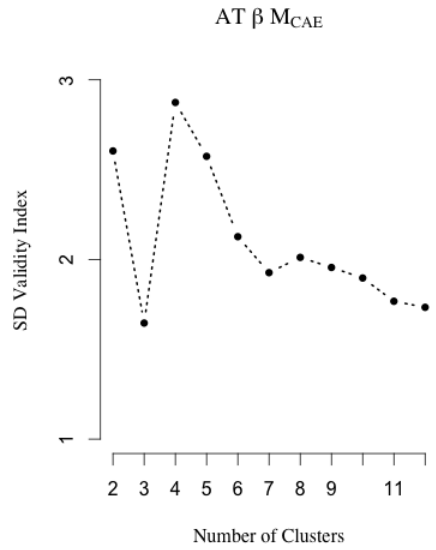


Figure 12. SD validity index values for the AT β M_{CAE} model.

PCA did not reduce the dimensions of either model. Table 17 details the dimensions and the values for K found by SD index analysis. Figure 11 shows the SD index values found for program AT α, while Figure 12 shows SD index values for AT β.

Table 18 shows summary statistics for models based on coupling on advice execution, and Table 19 shows *p*-values for Wilcoxon rank sum testing. When comparing models AT α M_{CAE} and AT β M_{CAE}, all metrics showed slightly better clusterings for program AT α. However, rank sum testing showed no significant improvement for the seeded program. Therefore, CAE does not likely provide good clustering results for locating advice and weaving interference. The granularity of the CAE metric was at the class-aspect or aspect-aspect level rather than the advice-advice, advice-method, and method-method level. Therefore, the coarse granularity hid the details required to show potential interference problems.

Table 18. Summary Statistics for AT Model M_{CAE} after 100 Runs of K-means++

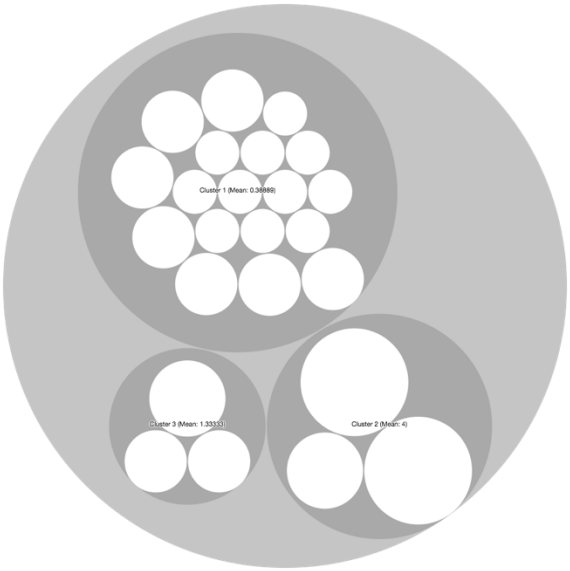
Measure	Model	Mean	Median	Std. Dev.
RS	AT α M_{CAE}	0.4864	0.4947	0.0266
	AT β M_{CAE}	0.4446	0.4437	0.0198
D	AT α M_{CAE}	0.5541	0.5774	0.0356
	AT β M_{CAE}	0.5371	0.5000	0.0388
DB	AT α M_{CAE}	1.1510	0.9707	0.2791
	AT β M_{CAE}	1.1679	1.0508	0.2326
SD	AT α M_{CAE}	2.9012	2.6136	0.5392
	AT β M_{CAE}	2.8078	2.6813	0.4060

Table 19. Wilcoxson Rank Sum Test *p*-Values for AT CAE Models

	A < B		**A > B**	
	RS	**D**	**DB**	**SD**
AT α M_{CAE}, AT β M_{CAE}	1.0000	0.9992	0.9995	0.3371

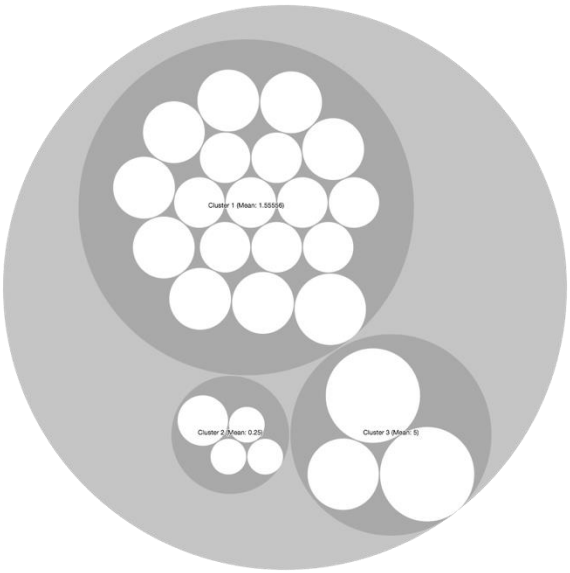
Figure 13 displays the visualizations of the best CAE clustering for program AT α. The maximum value for CAE was equal to the number of aspects in the system. An aspect may affect a module many times, but it counts only once toward the CAE total. This masks the impact of potential interference within a system. For program AT α (with 8 aspects), the highest-ranked cluster had classes AspectTetris (CAE=5) and Gui/TetrisGUI (CAE=5), and aspect Aspects/Highscore/Counter (CAE=2).

Figure 14 shows the best CAE clustering for program AT β. This clustering produced an increased mean for each cluster, but the SeedAspect (CAE=1) appeared in the cluster with the lowest mean. This suggested that introducing an aspect with high ICP had little effect on CAE modeling, and agreed with the summary statistics suggesting that the CAE model includes only nominal improvements for the modified program.



Model AT α M_{CAE}

Figure 13. Visualizations of Best CAE Clustering for Program AT α.



Model AT β M_{CAE}

Figure 14. Visualization of the Best CAE Clustering for Program AT β.

AJHotDraw Results

AJHD with Model M_{IP} (Interference Potential)

Following compilation, program AJHD α consisted of 3,953 methods and advices, resulting in 3,953 × 3,953 pattern matrix AJHD α M_{IP}. Program AJHD α contained 22 unique values for the IP metric. Compilation of the AJHD β program resulted in 4,037 objects, and the 4,037 × 4,037 AJHD β M_{IP} pattern matrix. Program AJHD β contained 25 unique values for the IP metric. Table 20 displays the range of K values tested for each program during SD analysis to determine K^*.

Table 20. Values of K Tested for AJHotDraw for Model M_{IP}

Model	K_{μ}	K_{min}	K_{max}
AJHD α M_{IP}	22	2	32
AJHD β M_{IP}	25	5	35

PCA of both versions of the AJHD program dramatically reduced dimensions in both cases. Pattern matrix AJHD α M_{IP} Π was of size 3,953 × 11, while pattern matrix AJHD β M_{IP}Π was of size 4,037 × 12. Table 21 displays the dimension and chosen value of K for each vector space model. Figure 15 and Figure 16 show the results of SD analysis, indicating a value of $K^* = 3$ for model AJHD α M_{IP}, $K^* = 29$ for model AJHD α M_{IP}Π, $K^* = 9$ for model AJHD β M_{IP}, and $K^* = 5$ for model AJHD β M_{IP} Π. The high value for K^* for model AJHD α M_{IP}Π seemed to be an outlier that could indicate a poor choice of K.

Table 21. Suggested Numbers of Clusters for AJHotDraw Model M_{IP}

Model	Dim	K^*
AJHD α M_{IP}	3953	3
AJHD α M_{IP}Π	11	29
AJHD β M_{IP}	4037	9
AJHD β M_{IP}Π	12	5

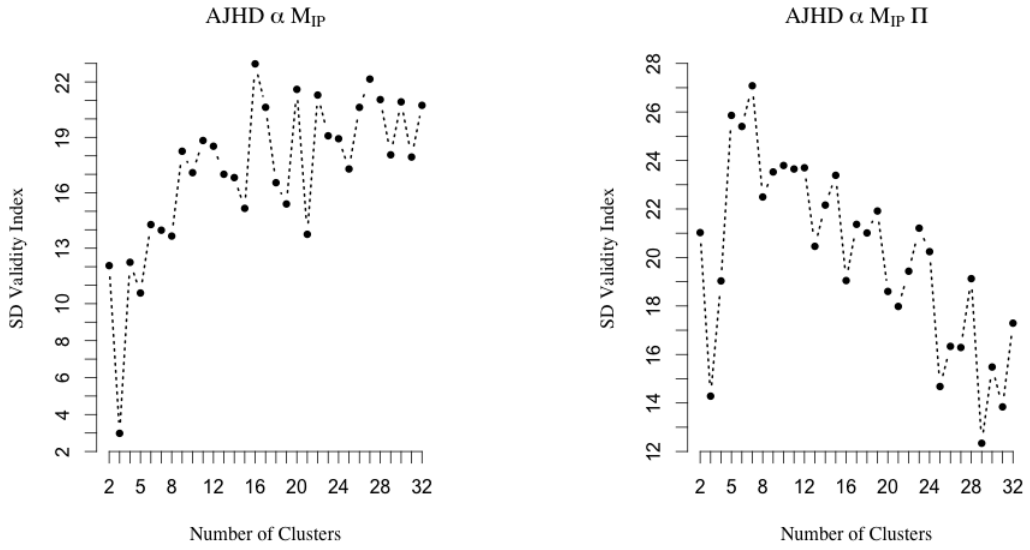


Figure 15. SD validity index values for the AJHD α M_{IP} and AJHD α M_{IP} Π models.

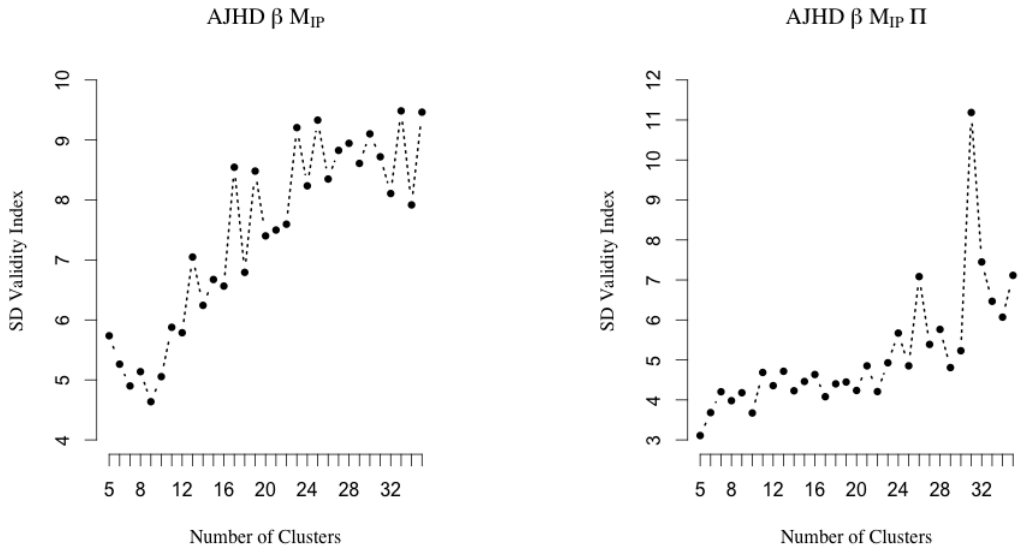


Figure 16. SD validity index values for the AJHD β M_{IP} and AJHD β M_{IP} Π models.

Using the suggested values for K, clustering was completed 100 times and summary statistics were collected. Table 22 displays the mean, median, and standard deviations for each metric and each model. Table 23 displays *p*-values resulting from Wilcoxon rank sum testing, showing that the DB metric produced no significant results.

Table 22. Summary Statistics for AJHD Model M_{IP} after 100 Runs of K-means++

Measure	Model	Mean	Median	Std. Dev.
RS	AJHD $\alpha\ M_{IP}$	0.0453	0.0459	0.0088
	AJHD $\alpha\ M_{IP}\Pi$	0.9157	0.9204	0.0173
	AJHD $\beta\ M_{IP}$	0.4045	0.4046	0.0055
	AJHD $\beta\ M_{IP}\Pi$	0.8579	0.8592	0.0122
D	AJHD $\alpha\ M_{IP}$	0.1499	0.1443	0.0387
	AJHD $\alpha\ M_{IP}\Pi$	0.0277	0.0273	0.0172
	AJHD $\beta\ M_{IP}$	0.1555	0.1543	0.0066
	AJHD $\beta\ M_{IP}\Pi$	0.0850	0.0970	0.0340
DB	AJHD $\alpha\ M_{IP}$	2.5259	2.4413	0.6527
	AJHD $\alpha\ M_{IP}\Pi$	0.6987	0.7042	0.0516
	AJHD $\beta\ M_{IP}$	2.5416	2.5124	0.3127
	AJHD $\beta\ M_{IP}\Pi$	0.9446	0.8747	0.2050
SD	AJHD $\alpha\ M_{IP}$	8.1638	8.2006	1.3024
	AJHD $\alpha\ M_{IP}\Pi$	8.5444	8.2301	2.3890
	AJHD $\beta\ M_{IP}$	3.9688	3.6904	1.1887
	AJHD $\beta\ M_{IP}\Pi$	1.8476	1.8021	0.2459

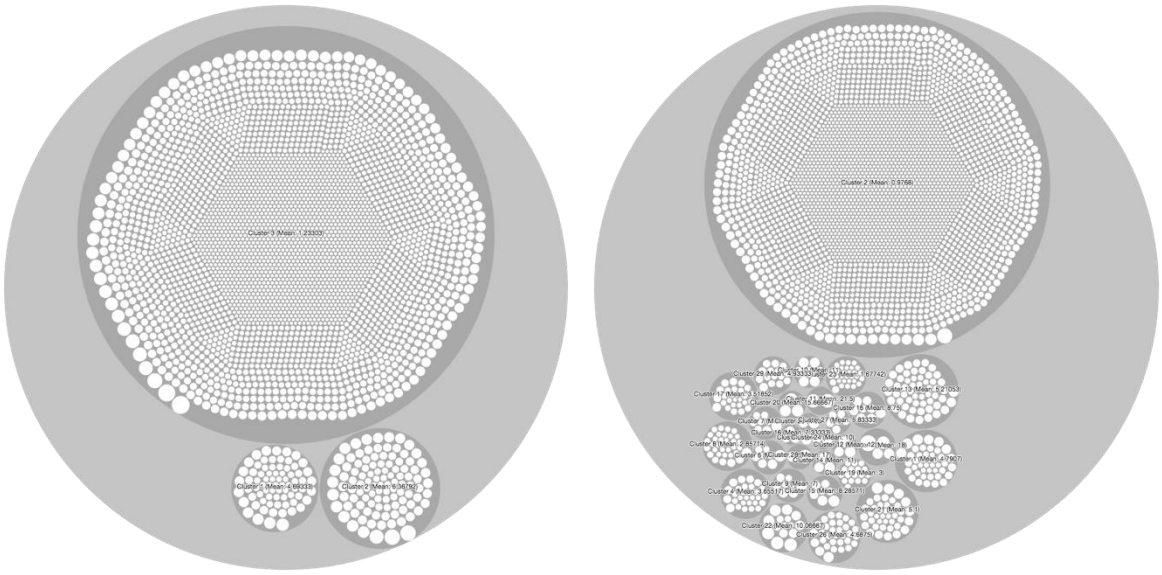
Table 23. Wilcoxon Rank Sum Test *p*-Values for AJHD IP Models

	A < B		A > B	
	RS	**D**	**DB**	**SD**
AJHD $\alpha\ M_{IP}$, AJHD $\alpha\ M_{IP}\Pi$	0.0000	0.0000	1.0000	0.8794
AJHD $\beta\ M_{IP}$, AJHD $\beta\ M_{IP}\Pi$	0.0000	0.0000	1.0000	0.0000
AJHD $\alpha\ M_{IP}$, AJHD $\beta\ M_{IP}$	0.0000	0.0000	0.9444	0.0000
AJHD $\alpha\ M_{IP}\ \Pi$, AJHD $\beta\ M_{IP}\ \Pi$	1.0000	0.5005	0.5005	0.0000

Metric mean scores seemed to indicate improved clusterings in both PCA reduced models, except by the SD metric. The RS and D metrics produced significant results for the AJHD $\alpha\ M_{IP}\Pi$ and AJHD $\beta\ M_{IP}\Pi$ models over the unreduced models, while the SD index showed a significant result for only the AJHD $\beta\ M_{IP}\Pi$ model. Model AJHD $\beta\ M_{IP}$ showed a significant result over AJHD $\alpha\ M_{IP}$ in the RS, D, and SD metrics. Only the SD metric showed a significant result for AJHD $\beta\ M_{IP}\Pi$ over AJHD $\alpha\ M_{IP}\Pi$.

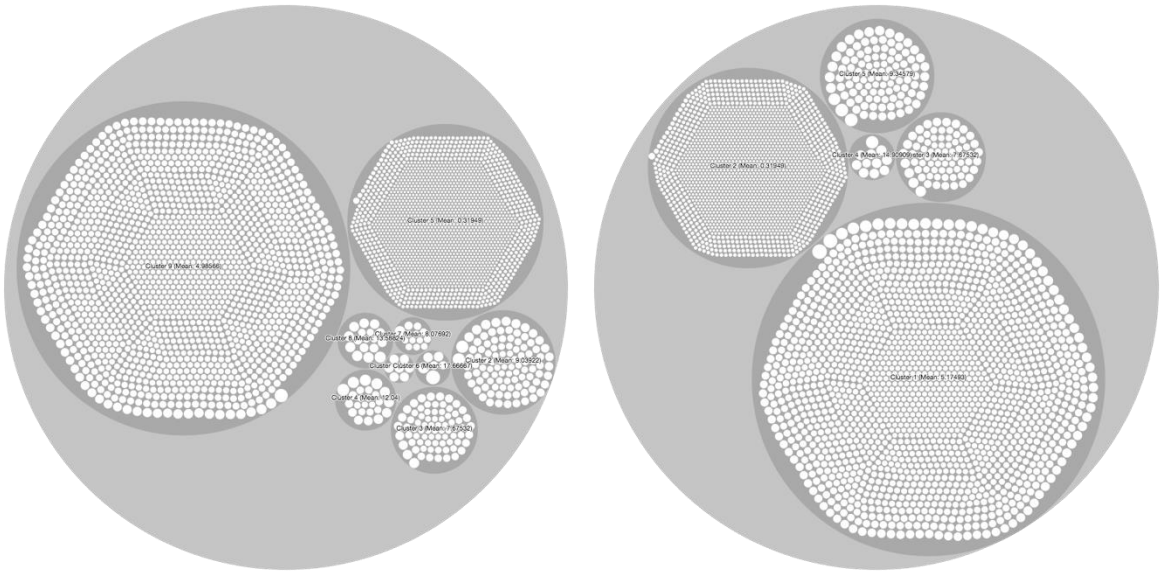
Mean RS values indicated that AJHD $\beta\ M_{IP}\Pi$ contained the most dissimilar clustering, while DB indicated that AJHD $\alpha\ M_{IP}\Pi$ was the most dissimilar. Note that the mean RS value for AJHD $\alpha\ M_{IP}$ was very low, indicating similar clusterings, and possibly suggesting that the chosen value for K was too low. The mean values for Dunn's index were lower than those in the AT IP results. This again shows that Dunn's index was quite susceptible to the diameter of the clusters, since the larger AJHD program

would contain larger-diameter clusters than the smaller AT program.



Model AJHD $\alpha\ M_{IP}$ Model AJHD $\alpha\ M_{IP}\Pi$

Figure 17. Visualizations for Program AJHD α IP Models.



Model AJHD $\beta\ M_{IP}$ Model AJHD $\beta\ M_{IP}\Pi$

Figure 18. Visualizations for Program AJHD β IP Models.

Visualizations of the best clustering (based on the RS value) appear in Figure 17 and Figure 18. Notice that the unmodified versions of AJHD produced a cluster with many objects, with a mean IP value of 1.233 in the unreduced model, and a mean IP

value of 0.977 in the reduced model. The seeded versions of the program showed that the large cluster from the unmodified version of the program split into multiple clusters in the modified version. This suggested that the seeded advices raised the IP value of many of the methods slightly, and this fact was detected by the clustering. The lowest-ranked cluster in both AJHD β clusterings contained a mean IP of 0.319, while the next cluster had mean IP values of around 5. The same phenomenon occurred in the IP models of the AT program, which demonstrates that the IP clustering method is scalable to large programs with similar behavior.

When looking more closely at the resulting clusters, an anomaly was noted. Unlike the AT program, the method with the highest IP value did not appear in the highest-ranked cluster in all cases. The org/jhotdraw/samples/javadraw/JavaDrawApp.createTools method had the highest IP value (IP=30 in the unmodified version and IP=33 in the modified version). Interestingly, this method appeared in the highest ranked cluster only in the AJHD β M_{IP} model. It appeared in the lowest ranked clusters for both the AJHD α M_{IP} and AJHD β $M_{IP}\Pi$ models, and a moderately ranked cluster for the AJHD α $M_{IP}\Pi$ model. This indicated that the clustering was less effective in pinpointing the method with the highest IP in the larger scale AJHD program than the smaller AT program. This may be further evidence that the chosen value of K was too low for some of the clusterings.

AJHD with Model M_{ICP} (Interference Causality Potential)

Because M_{ICP} came directly from the transposition of M_{IP}, pattern matrices were of the same sizes: 3,953 × 3,953 for AJHD α, and 4,037 × 4,037 for AJHD β. Program AJHD α produced 33 unique values for the ICP metric, while AJHD β produced 37

unique values. Table 24 details the values of K used to determine K^*.

Table 24. Values of K Tested for AJHotDraw Model M_{ICP}

Model	K_μ	K_{min}	K_{max}
AJHD α M_{ICP}	33	2	43
AJHD α M_{ICP} Π	33	13	43
AJHD β M_{ICP}	37	17	47

PCA reduced the dimension of both models. AJHD α M_{ICP}Π included only 11 dimensions, while model AJHD β M_{ICP}Π included only 12 dimensions. Table 25 shows the dimensionality reduction, and the chosen values for K in each model. Note that model AJHD α M_{ICP} suggested a small number of clusters, while other models indicated a value for K^* between 28 and 30. This again seemed to suggest a problem with the chosen value for K. Figure 19 shows the SD analysis for AJHD α, while Figure 20 shows SD analysis plots for AJHD β.

Table 25. Suggested Numbers of Clusters for AJHotDraw Model M_{ICP}

Model	Dim	K^*
AJHD α M_{ICP}	3953	3
AJHD α M_{ICP}Π	11	28
AJHD β M_{ICP}	4037	30
AJHD β M_{ICP}Π	12	24

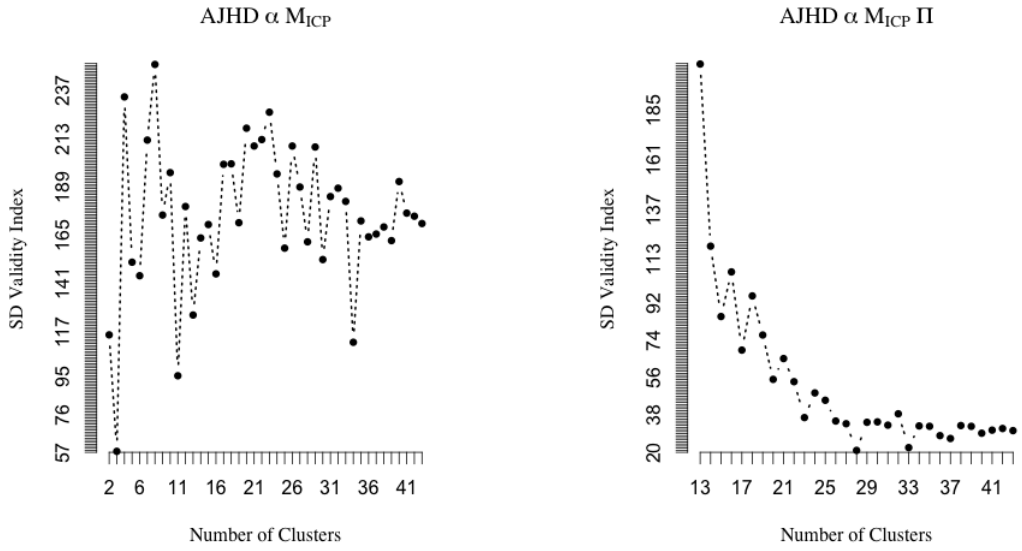


Figure 19. SD validity index values for the AJHD α M_{ICP} and AJHD α M_{ICP} Π models.

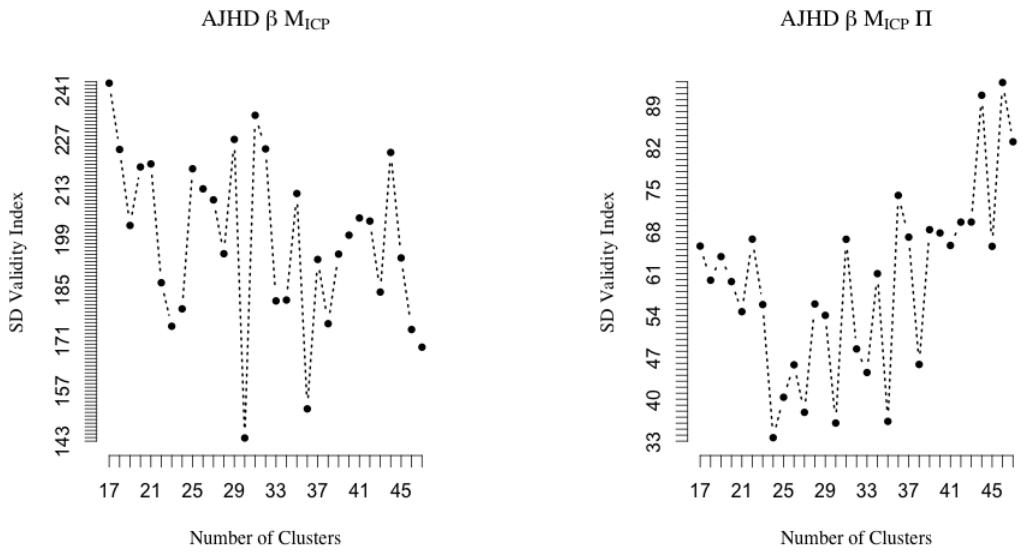


Figure 20. SD validity index values for the AJHD β M_{ICP} and AJHD β M_{ICP} Π models.

Table 26 shows summary statistics following 100 runs of AJHD program clustering. Table 27 includes Wilcoxon rank sum test *p*-values for comparisons. The mean RS score for AJHD $\alpha\ M_{ICP}$ was significantly lower than RS scores for the other three models, suggesting that the clusters are very similar, likely because of the low number of clusters. The Dunn index indicated a different result, suggesting AJHD $\alpha\ M_{ICP}$ produced the best cluster configuration. This again implied that Dunn's index could produce different values depending upon the number and the diameter of clusters. Both the DB and SD metrics indicated that AJHD $\alpha\ M_{ICP}\Pi$ produced the best clustering.

Rank sum testing showed significant shifts in RS, DB, and SD from the AJHD $\alpha\ M_{ICP}$ model to its PCA-reduced version. RS and DB showed significant results when comparing AJHD $\beta\ M_{ICP}$ to its PCA-reduced version. Note that only RS produced a significant result from the non-reduced AJHD α model to the AJHD β model, while RS and D showed significant shifts from the reduced AJHD α model to the AJHD β model. These mixed results seemed to further suggest a problem with the chosen value of K.

Table 26. Summary Statistics for AJHD Model M_{ICP} after 100 Runs of K-means++

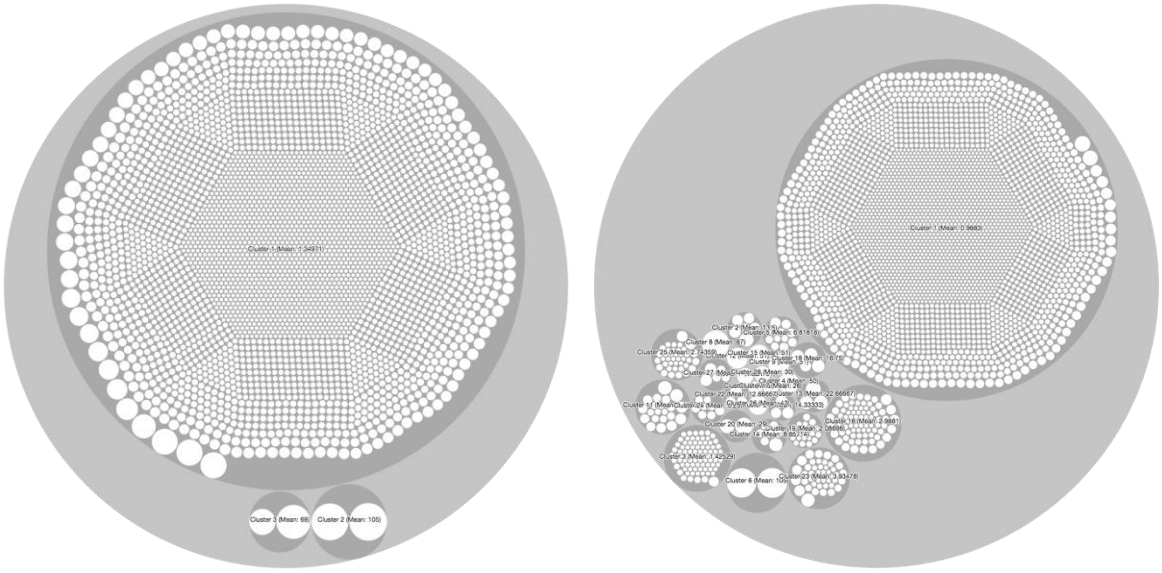
Measure	Model	Mean	Median	Std. Dev.
RS	AJHD α M_{ICP}	0.0487	0.0519	0.0080
	AJHD α $M_{ICP}\Pi$	0.9433	0.9442	0.0039
	AJHD β M_{ICP}	0.6092	0.6090	0.0018
	AJHD β $M_{ICP}\Pi$	0.9886	0.9888	0.0009
D	AJHD α M_{ICP}	0.2801	0.2148	0.1733
	AJHD α $M_{ICP}\Pi$	0.0071	0.0063	0.0045
	AJHD β M_{ICP}	0.1395	0.1367	0.0099
	AJHD β $M_{ICP}\Pi$	0.0098	0.0090	0.0066
DB	AJHD α M_{ICP}	1.8461	1.8156	0.8192
	AJHD α $M_{ICP}\Pi$	0.5770	0.5738	0.0610
	AJHD β M_{ICP}	2.2853	2.2822	0.2501
	AJHD β $M_{ICP}\Pi$	0.6264	0.6171	0.0789
SD	AJHD α M_{ICP}	20.8518	16.9494	6.7815
	AJHD α $M_{ICP}\Pi$	16.8257	16.9193	4.4700
	AJHD β M_{ICP}	29.3368	28.8259	4.0288
	AJHD β $M_{ICP}\Pi$	31.1787	26.9978	13.1705

Table 27. Wilcoxon Rank Sum Test *p*-Values for AJHD ICP Models

	A < B		A > B	
	RS	**D**	**DB**	**SD**
AJHD α M_{ICP}, AJHD α $M_{ICP}\Pi$	0.0000	1.0000	0.0000	0.0001
AJHD β M_{ICP}, AJHD β $M_{ICP}\Pi$	0.0000	1.0000	0.0000	0.1139
AJHD α M_{ICP}, AJHD β M_{ICP}	0.0000	1.0000	1.0000	1.0000
AJHD α M_{ICP} Π, AJHD β M_{ICP} Π	0.0000	0.0002	1.0000	1.0000

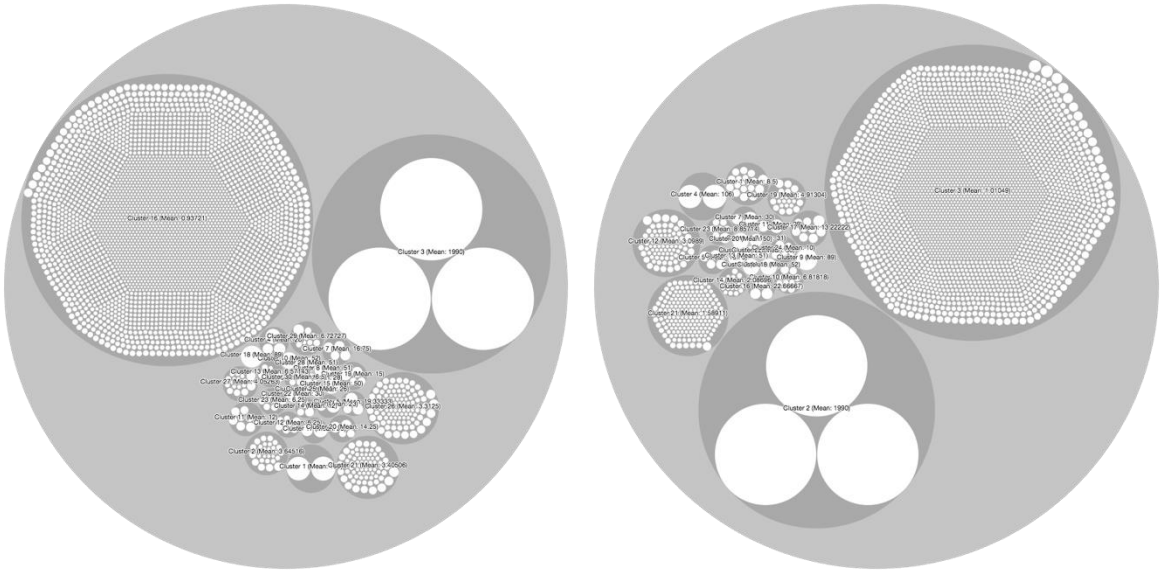
Figure 21 shows the clusterings that produced the highest RS value for AJHD α. The highest-ranked cluster in both the reduced and unreduced models contained the method with the highest ICP: org/jhotdraw/framework/FigureEnumeration.hasNextFigure (ICP=107).

Figure 22 shows clusterings that produced the highest RS values for AJHD β. Note the dramatic increase in the size of the highest-ranked cluster. In both cases, this cluster contained the three bytecode methods that resulted from org/jhotdraw/SeedAspect: the before and after advice, and the aspectOf method, each with an ICP value of 1,990. This shows that, like the small-scale AspectTetris program, the clustering algorithm with the ICP metric was able to detect and identify the methods with the potential to cause interference problems within a program.



Model AJHD α M_{ICP} Model AJHD α $M_{ICP}\Pi$

Figure 21. Visualizations for Program AJHD α for ICP Models.



Model AJHD β M_{ICP} Model AJHD β $M_{ICP}\Pi$

Figure 22. Visualizations for Program AJHD β for ICP Models.

AJHD with Model M_{CAE} (Coupling on Advice Execution)

AJHD α included 396 classes and 31 aspects, resulting in a pattern matrix of size 427×31. AJHD β included 407 classes and 32 aspects, resulting in a pattern matrix of

size 439 × 32. There were 5 unique values for CAE in program AJHD α, and 6 unique values in program AJHD β. Table 28 details the values of K tested using SD analysis to determine the value of K^* during the clustering phase.

Table 28. Values of K Tested for AJHotDraw Model M_{CAE}

Model	K_{μ}	K_{min}	K_{max}
AJHD α M_{CAE}	5	2	15
AJHD β M_{CAE}	6	2	16

PCA reduced the dimensions of the pattern matrices to 427 × 6 for AJHD α $M_{CAE}\Pi$ and 439 × 7 for AJHD β $M_{CAE}\Pi$. Table 29 shows the suggested values of K determined by SD analysis, ranging between 2 and 10. Figure 23 and Figure 24 plot the SD results for each of the models tested.

Table 29. Suggested Numbers of Clusters for AJHotDraw Model M_{CAE}

Model	Dim	K^*
AJHD α M_{CAE}	31	8
AJHD α $M_{CAE}\Pi$	6	10
AJHD β M_{CAE}	32	2
AJHD β $M_{CAE}\Pi$	7	6

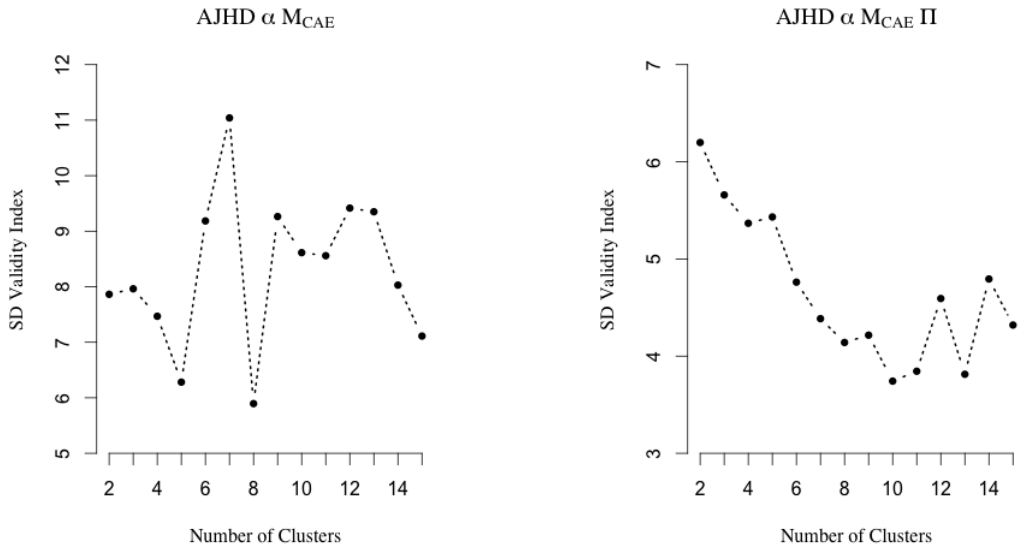


Figure 23. SD validity index values for the AJHD α M_{CAE} and AJHD α M_{CAE} Π models.

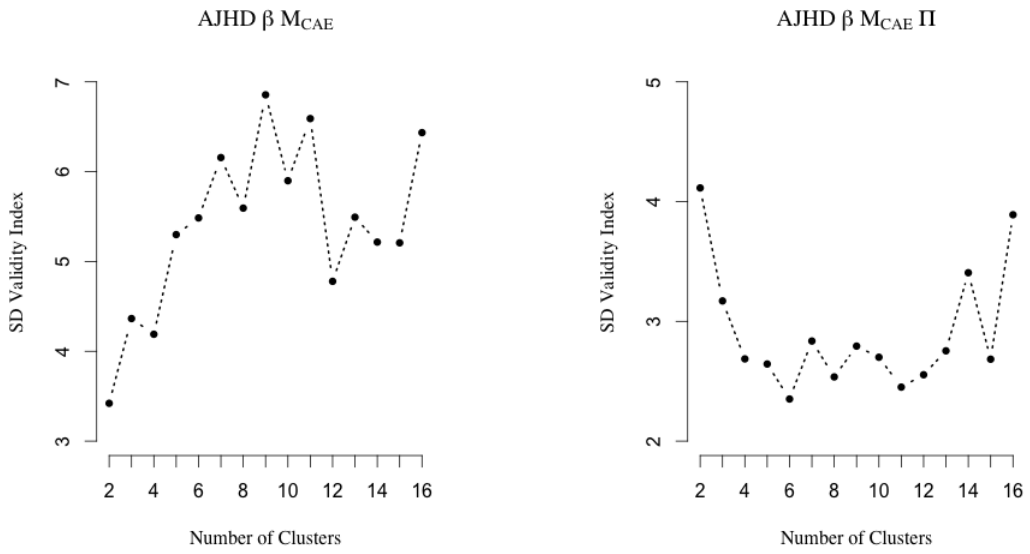


Figure 24. SD validity index values for the AJHD β M_{CAE} and AJHD β M_{CAE} Π models.

Table 30 displays summary statistics following 100 runs of the clustering algorithm. The RS index and DB index both indicated an advantage for AJHD $\alpha\ M_{CAE}\Pi$—suggesting that it had the most dissimilar clusters. Metric D gave only a slight advantage to model AJHD $\alpha\ M_{CAE}$, while SD gave a slight advantage to AJHD $\beta\ M_{CAE}\Pi$. These results seemed to indicate an unremarkable effect on the CAE metric when adding the seeded potential for interference.

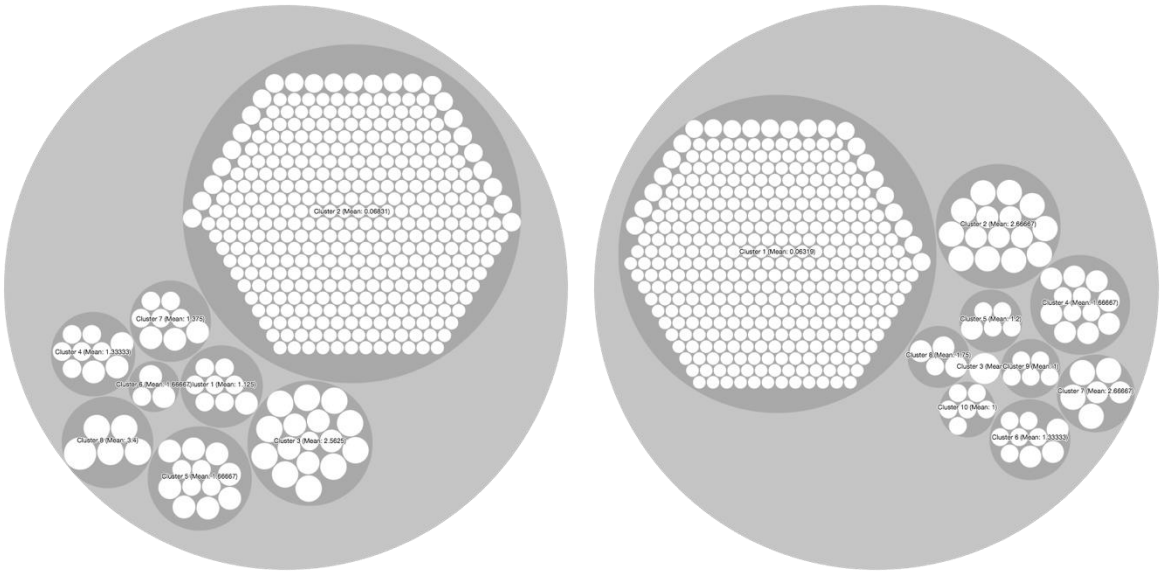
Table 30. Summary Statistics for AJHD Model M_{CAE} after 100 Runs of K-means++

Measure	Model	Mean	Median	Std. Dev.
RS	AJHD $\alpha\ M_{CAE}$	0.5705	0.5727	0.0196
	AJHD $\alpha\ M_{CAE}\Pi$	0.9322	0.9291	0.0173
	AJHD $\beta\ M_{CAE}$	0.2439	0.2074	0.0598
	AJHD $\beta\ M_{CAE}\Pi$	0.7884	0.8287	0.1273
D	AJHD $\alpha\ M_{CAE}$	0.5428	0.5000	0.0553
	AJHD $\alpha\ M_{CAE}\Pi$	0.4523	0.4709	0.0693
	AJHD $\beta\ M_{CAE}$	0.4202	0.4472	0.0339
	AJHD $\beta\ M_{CAE}\Pi$	0.3097	0.2862	0.0756
DB	AJHD $\alpha\ M_{CAE}$	1.2920	1.2503	0.1767
	AJHD $\alpha\ M_{CAE}\Pi$	0.4578	0.4409	0.0524
	AJHD $\beta\ M_{CAE}$	1.1253	1.1747	0.1416
	AJHD $\beta\ M_{CAE}\Pi$	0.6289	0.5854	0.1067
SD	AJHD $\alpha\ M_{CAE}$	4.5087	4.4479	0.7666
	AJHD $\alpha\ M_{CAE}\Pi$	3.1828	2.8854	0.6783
	AJHD $\beta\ M_{CAE}$	2.7939	2.9368	0.2776
	AJHD $\beta\ M_{CAE}\Pi$	2.2500	2.1674	0.3120

Table 31 displays *p*-values for Wilcoxon rank sum testing. The RS index showed significant results between the unreduced and reduced models, but no significant results between the unmodified and the modified versions of the program. Dunn's index produced no significant results. The DB metric showed significant results on all except the unmodified reduced matrix to the modified reduced matrix. The SD index showed significant improvements to all four clustering hypotheses.

Table 31. Wilcoxon Rank Sum Test *p*-Values for AJHD CAE Models

	A < B		**A > B**	
	RS	**D**	**DB**	**SD**
AJHD α M_{CAE}, AJHD α $M_{CAE}\Pi$	0.0000	1.0000	0.0000	0.0000
AJHD β M_{CAE}, AJHD β $M_{CAE}\Pi$	0.0000	1.0000	0.0000	0.0000
AJHD α M_{CAE}, AJHD β M_{CAE}	1.0000	1.0000	0.0000	0.0000
AJHD α M_{CAE} Π, AJHD β M_{CAE} Π	1.0000	1.0000	1.0000	0.0000

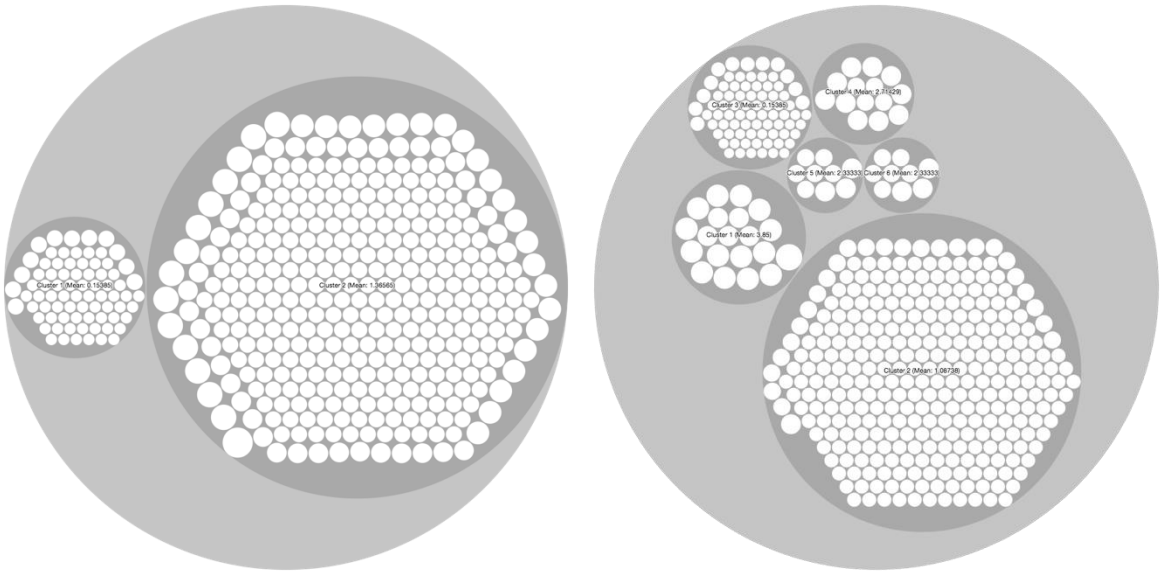


Model AJHD α M_{CAE} Model AJHD α $M_{CAE}\Pi$

Figure 25. Visualizations for Program AJHD α for CAE Models.

When comparing CAE clusterings visually, no remarkable changes were noted between AJHD α (Figure 25) and AJHD β (Figure 26). The highest-ranked clusters in all cases contained the objects with the highest CAE: org/jhotdraw/standard/AbstractCommand (CAE=5 in AJHD α and CAE=6 in AJHD β).

This was more difficult to discern in the visualization produced by AJHD $\beta\ M_{CAE}$, because the cluster with the largest number of objects had the highest rank. This suggested that a larger number of clusters would have produced a more remarkable result. In addition, the AbstractCommand class contained approximately 125 bytecode methods. Because the metric hid the details of the interaction, it was unable to pinpoint the advice or method with the highest potential for interference.



Model AJHD $\beta\ M_{CAE}$ Model AJHD $\beta\ M_{CAE}\Pi$

Figure 26. Visualizations for Program AJHD β for CAE Models.

Summary of Results

This section discusses overall results of this work, and addresses each of the research questions posed. Evidence from the data collected and presented in this chapter provides the foundation for this section. Table 32 summarizes statistical improvements shown throughout this chapter, and is used here to discuss the overall results.

Overall Impressions

The results presented in this chapter produced three interesting observations. First, determining the value of K by using the SD index produced variable results, especially in

the large program. For example, both Table 21 and Table 25 noted both high and low values for K in the same vector space. This could indicate that the range of K values tested needed to be adjusted to determine if a better value for K existed. Multiple runs of the SD index analysis could produce different values for K. SD index values could vary because the *k*-means algorithm assigns objects to a given cluster center by attempting to locate the lowest within cluster sum of squares. The algorithm repeatedly reassigns objects to other clusters until no lower clustering can be found. Thus, the nature of the *k*-means assignment step could produce different clusterings and different values for the SD index. Therefore, because of the range of values tested and the variability of the SD index, the values of K found in this study may not be the best overall values for K.

Table 32. Summary of Statistics Showing Significant Improvements

	RS	**D**	**DB**	**SD**
AT $\alpha\ M_{IP}$, AT $\alpha\ M_{IP}\Pi$	X		X	X
AT $\beta\ M_{IP}$, AT $\beta\ M_{IP}\Pi$	X		X	X
AT $\alpha\ M_{IP}$, AT $\beta\ M_{IP}$	X		X	X
AT $\alpha\ M_{IP}\ \Pi$, AT $\beta\ M_{IP}\ \Pi$	X		X	X
AT $\alpha\ M_{ICP}$, AT $\alpha\ M_{ICP}\Pi$	X		X	
AT $\beta\ M_{ICP}$, AT $\beta\ M_{ICP}\Pi$	X	X	X	X
AT $\alpha\ M_{ICP}$, AT $\beta\ M_{ICP}$	X		X	X
AT $\alpha\ M_{ICP}\ \Pi$, AT $\beta\ M_{ICP}\ \Pi$	X	X	X	X
AT $\alpha\ M_{CAE}$, AT $\beta\ M_{CAE}$				
AJHD $\alpha\ M_{IP}$, AJHD $\alpha\ M_{IP}\Pi$	X	X		
AJHD $\beta\ M_{IP}$, AJHD $\beta\ M_{IP}\Pi$	X	X		X
AJHD $\alpha\ M_{IP}$, AJHD $\beta\ M_{IP}$	X	X		X
AJHD $\alpha\ M_{IP}\ \Pi$, AJHD $\beta\ M_{IP}\ \Pi$				X
AJHD $\alpha\ M_{ICP}$, AJHD $\alpha\ M_{ICP}\Pi$	X		X	X
AJHD $\beta\ M_{ICP}$, AJHD $\beta\ M_{ICP}\Pi$	X		X	
AJHD $\alpha\ M_{ICP}$, AJHD $\beta\ M_{ICP}$	X			
AJHD $\alpha\ M_{ICP}\ \Pi$, AJHD $\beta\ M_{ICP}\ \Pi$	X	X		
AJHD $\alpha\ M_{CAE}$, AJHD $\alpha\ M_{CAE}\Pi$	X		X	X
AJHD $\beta\ M_{CAE}$, AJHD $\beta\ M_{CAE}\Pi$	X		X	X
AJHD $\alpha\ M_{CAE}$, AJHD $\beta\ M_{CAE}$			X	X
AJHD $\alpha\ M_{CAE}\ \Pi$, AJHD $\beta\ M_{CAE}\ \Pi$				X

Second, the use of Dunn's index as a metric for comparing clusters seems inadequate—especially when the values of K differed among the clusterings compared. Table 32 shows that Dunn's index produced significant results in approximately 29% of the cases tested (6 of 21), while the other metrics showed significant results at a much

higher rate (approximately 81% for RS, 71% for SD, and 62% for DB). Because Dunn's index relied heavily on cluster diameter, the value of K played an important part in the metric. Thus, in analyses where K differs among clusterings being compared, Dunn's index is not a good choice to show validity among the groups of clusterings.

Third, PCA reduction was successful in 28 of the 40 cases tested (~70%) (Table 32). The CAE metric produced the highest rate of significant results for PCA reduction (6 of 8 or 75%). However, the CAE vector space model for AspectTetris resulted in no dimensionality reductions, which skews the result. Both the IP and ICP models showed significant results in 11 of 16 (~69%) of the cases tested. Therefore, the use of PCA reduction may help to increase the efficiency of this type of static analysis.

Evaluation of the IP and ICP Metrics

The first research question asked whether new fine-grained metrics could adequately describe the potential for aspect interference. The two new metrics defined in this study were the interference potential of an object (IP), and the interference causality potential for an object (ICP). IP counted the number of methods or advices called by the given method. ICP counted the number of methods or advices that interacted with the given method. Because no other fine-grained aspect coupling metrics existed, the study compared results to the CAE (coupling on advice execution) metric, which described the number of aspects that affected a given class or aspect.

Evidence presented in this study indicated that IP and ICP were adequate for describing method-method, advice-method, and advice-advice interactions. Across both programs tested, each metric produced significant results a total of 42 of 64 times (65%) (Table 32). When considering improvements from the unmodified version of the program

to the modified version, each metric produced significant results 10 of 16 times (63%) (Table 32). The similar results may reflect the two metrics' complementary nature. When viewing the resulting clusters, introducing SeedAspect to each program produced more meaningful clusterings with the ICP metric. While IP clusterings seemed to divide the largest cluster in the unmodified programs into smaller clusters in the modified programs, the ICP clusterings showed dramatic increases in the highest-ranked clusters. This highlights the difficulty of ensuring a high IP without manually creating a method that invokes many other methods. Thus, while results showed that both ICP and IP changed in a program with an increased possibility for aspect interference, ICP showed a more dramatic effect.

CAE clusterings showed improvements in 45% (9 of 20) of the cases tested, but only 3 of 12 (25%) of cases from the unmodified version to the modified version of the programs (Table 32). This agreed with the assessment that a coarsely grained metric such as CAE was inadequate for locating potential interference, and strengthened the case for the IP and ICP metrics.

The Use of Clustering Analysis to Detect Aspect Interference Potential

The second research question asked whether clustering analysis was a viable tool for detecting potential aspect interference within an aspect-oriented program. The clusters resulting from both the IP and ICP metrics show improved dissimilarity between the unmodified and the modified versions of the program 20 of 32 times (63%) (Table 32). This suggested that the modified versions of the program containing increased interference potential successfully affected the clusterings. Viewing clusters visually showed marked changes in both the IP and ICP clusterings for each application,

indicating that the potential for aspect interference was detectable by clustering analysis. The follow-up to this question asked whether clustering could locate both the potential to cause interference and the methods or advices that may be the victims of interference problems. Analysis of the clusters showed that the object with the highest ICP value always appeared in the highest-ranked cluster. However, the object with the highest IP values did not necessarily appear in the highest-ranked cluster. While evidence indicated that clustering detected both items with the potential to interfere (IP) and the potential to be interfered with (ICP), clustering revealed objects with high ICP more directly. However, recall that the IP metric was more difficult to test because it would require many method calls from a single method. Therefore, further testing of the IP metric will be required to ensure that clustering was viable for methods or advices with the highest interference potential.

The third research question dealt with the study's scalability, asking whether clustering analysis scaled to a larger-scale program. The summary in Table 32 shows statistically significant results in 25 of 36 cases (69%) for the AT program. AJHD, the larger program, showed significant results in 26 of 48 cases (54%). When excluding CAE results, program AT produced 25 of 32 (78%) significant outcomes, while program AJHD produced only 17 of 32 (53%) significant outcomes. This suggests that the larger program had lower-quality clusterings than the smaller program. Recall that the value for K may have been poor in some cases due to the variability in the SD index. This phenomenon was most clearly evident in the AJHD program, which produced outlier values for K in both the ICP and IP models. AJHD model clustering metrics may have trended down in cases where K was not optimal, resulting in fewer significant results.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

This dissertation sought to collect fine-grained coupling metrics from the woven bytecode of an AspectJ program and use them to pinpoint areas where aspect interference problems may occur. Data collected using two new fine-grained aspect coupling metrics—interference potential and interference causality potential—as well as a classic coarse-grained aspect coupling metric—coupling on advice execution—provided the framework for vector space models that fed into the *k*-means++ clustering algorithm. Collecting statistics and analyzing resulting clusters led to the following conclusions.

First, the clustering methodology used in this work needs improvement. The results indicated that the optimal value for K might not have been located in all cases. Therefore, using another method to locate a better value for K is recommended. One possibility, noted in the results, was that the range of K values used in SD analysis was not broad enough. This study based this range on the number of unique values for the metric described by the vector space model. Other methods for finding a range of K values to test may improve the clustering results. In addition, the results showed the irregular nature of the SD index for determining K. Because of the fluctuations in the SD index, using a different technique may also improve results.

Second, the use of a fine-grained metric over a coarsely grained metric was essential for pinpointing potential advice and weaving interference. Results showed that

both the IP and ICP metric models produced higher rates of significant results than CAE metric models. This was because coupling at the class/aspect level excluded essential details required for determining precisely where an interference problem could manifest or originate. Using finer-grained metrics like IP and ICP allowed a much more accurate picture of where interference problems might exist. Though this study suggested a more meaningful result in ICP clusterings, IP clusterings clearly changed as a result of increased interference potential. This lent to the credibility of both IP and ICP as aspect coupling metrics.

The study of the IP metric was limited by the difficulty of creating a method with high IP. To effectively raise the IP of a single method, one must alter the code of the method to call many methods. Given the size of the programs used in this work, altering methods in this manner was not performed. This limitation also presents a weakness in the results of the IP clustering. Future research should perform a more thorough examination of IP clustering models and ensure that a high IP method exists in the studied system.

Implications

This was the first known study to attempt to locate aspect interference via code analysis since D'Ursi, Cavallaro, and Monga (2007). D'Ursi, Cavallaro, and Monga (2007) used simple program slicing as a method for locating interference problems in code, but found it ineffective. Still, these authors maintained that other static code analysis techniques had value. The current study confirms the implication that locating potential interference problems with static code analysis is possible (D'Ursi et al., 2007).

This study showed that static bytecode analysis using clustering could pinpoint

methods or advices with high ICP. The clustering analysis also detected changes in IP clusterings when methods having high ICP were introduced into the code. IP clusterings were less conclusive than ICP clusterings because of the limitations of introducing a method with a high IP. Despite this limitation, this work provided a solid foundation for future research using clustering analysis to locate potential interference problems.

This study also showed that fine-grained metrics like IP and ICP produced more meaningful results that older, coarsely grained aspect coupling metrics. The IP and ICP metrics represented the potential for both weaving interference (an advice's effect on class code) and advice interference (an advice's effect on other advices). The CAE metric showed very small changes in clusterings, especially in the smaller program. This indicated that coarse-grained metrics hid the details of coupling required to show advice and weaving interference. Therefore, fine-grained metrics were essential for completing this work because detecting potential interference problems at the advice and method levels was otherwise impossible.

Finally, this was the first known study to introduce a zoomable visualization technique for presenting clusters related to code. Previous studies such as Dietrich et al. (2008) and Cassell et al. (2011) used dependency graphs to display clustering results. A zoomable format shifted the viewer's focus to individual clusters that involved high interference potential, and opened the door to future studies that use this technique.

Recommendations

Previous sections detailed future research gleaned from the results of this study. While these opportunities are important, they involve a narrow scope. Other research opportunities with a broader scope also exist.

Because this study analyzed compiled AspectJ programs rather than source code, several interesting features were noted. The AspectJ compiler, to accommodate the different types of advice, sometimes split a single class into multiple bytecode classes. Only one study tracked the way AspectJ weaves code at compile-time (Hilsdale & Hugunin, 2004). Because of this, several prospects for future research exist. First, modernizing the work of Hilsdale and Hugunin (2004) to discover new join point shadows in Java bytecode is worthwhile. This would be useful because the IP and ICP metrics presented in the current study came directly from bytecode. A better understanding of the compilation and weaving process would allow a mapping between the IP and ICP bytecode metrics and the original source code. Using IP and ICP as source code metrics would likely make more sense to programmers, and would allow for more meaningful visualizations that clear out additional objects added by AspectJ compilation. Second, no known studies have determined whether the conversion of source code to bytecode by the AspectJ compiler increases the chances of aspect interference. Some of the decisions made during compilation and weaving may unintentionally increase the potential for aspect interference. Thus, studying the effects of the compiler itself on aspect interference is worthy of future study.

Another broad area for future research involves the use of clustering analysis to locate potential interference problems. While previous sections noted the need to improve upon the clustering techniques presented in this study, other opportunities exist. This dissertation showed that vector space modeling provided additional detail resulting in improvements in clusterings. Determining whether vector space models provided the best clusterings remains unseen. Thus, comparative studies with simple metric-based

clustering using *k*-means, hierarchical, or model-based clustering techniques using the IP and ICP metrics is worth investigation.

One should also note that the IP and ICP metrics were simple counts of interactions among methods and advices. Using simple counts in any study increases the difficulty of making comparisons because they depend so heavily on the program size. Therefore, future research may suggest modification of these metrics to make them easier to compare across programs. For example, by converting the metrics to a ratio of the current metric value to the maximum metric value in a system, one could then compare percentages between programs.

This study was limited to detecting advice and weaving interference as defined by Tian, Cooper, Zhang, and Liu (2010), and did not account for introduction interference. Advice interference involves problems resulting from advices that interact at a common join point, while weaving interference may result in violations of the expected system structure and flow. Introduction interference occurs when an aspect adds or deletes features from the program. The introduced metrics were unable to detect introduction interference. Thus, new metrics and techniques that account for introduction interference is another useful area of exploration.

Finally, this research introduced a zoomable visualization technique for illustrating clusters that was not studied thoroughly. While it would seem an effective way for programmers to analyze potential aspect interference, empirical study is necessary. Performing a human-computer interaction analysis of the clustering visualizations presented in this dissertation will help understand how programmers interact with and understand the clustering results, and provide insight into possible

improvements.

Summary

This dissertation discussed the history of object-oriented design, noting that the focus on functional decomposition design methodologies resulted in non-functional elements to be scattered throughout class code. Aspect-oriented programming (AOP) introduced a way to extract these scattered elements, known as crosscutting concerns, into a single encapsulation that wove concerns back into the class code where needed. The encapsulation, called an aspect, included point cuts that defined weaving locations generically and advice that was inserted before, after, or around a weaving location. Since point cuts are generic, they could cause semantic problems that were not readily apparent to programmers. At runtime, woven code may exhibit unexpected changes in the intended flow of the program. These unexpected changes were termed aspect interference.

This study sought to address the problem that computer practitioners had no way to fully conceptualize aspect interference that may exist in a program under development. Most interference research wanted to shield a program from the problems created when aspect interference occurs. Design-time techniques attempted to prevent interference problems before implementation began by formally defining a program to be interference-free. Other techniques required specific definitions within the code that prevented interference at execution time. All of these techniques required programmers to understand locations within the system that were most vulnerable to interference. Thus, the goal of this study was to give programmers a better understanding of potential interference by static code analysis.

A review of the literature showed that the definition of aspect interference has changed over time from a broad definition of interactions between aspects, to one that includes more details. Defining advice interference, weaving interference, and introduction interference gave a stronger foundation for studying interference problems. Introduction interference was left to future research because it involved an aspect that added or removed elements from a program and was beyond the scope of this study. Both advice and weaving interference dealt with interacting elements at the advice-advice, advice-method, and method-method granularities.

Such fine-grained interactions required fine-grained metrics. A review of existing aspect coupling metrics found that few metrics existed with such a small granularity. A study by Zhao (2004) mentioned advice-method and method-method interactions as separate metrics. However, this did not allow for the level of detail required in this work. Therefore, two new metrics—the interference potential of an object (IP) and the interference causality potential for an object (ICP)—provided a way to count all items (methods and advices) that call or were called by a method or advice.

Research in program refactoring showed that clustering analysis techniques had promise. In the OOP world, clustering was used to determine possible refactoring opportunities. In AOP research, clustering was used to locate potential crosscutting concerns in class code, allowing programmers to pull them into aspects. No existing studies used clustering analysis to locate areas of an existing AOP program where aspect interference problems might occur.

Creating a clustering technique for interference analysis required several steps. First, because the study was interested in potential interference after weaving, compiling

the program to bytecode was necessary. Following conversion, a Ruby parser reviewed the bytecode to identify objects (classes, aspects, methods, and advices) and the interactions among them. The parsing results were fashioned into pattern matrices: one denoting the IP/ICP of each object, and another denoting the coupling on advice execution (CAE). PCA reduction removed excess noise in the data, and reduced the number of dimensions in most cases. To determine the value of K for use in the *k*-means++ clustering algorithm, the study reviewed SD index results across several potential values of K. K was chosen where the SD index was minimized. The *k*-means++ algorithm was run 100 times for each model with the chosen K, and statistics and clusterings were retained. The R-Squared (RS), Dunn, Davies-Bouldin, and SD indexes were collected for each run, and the mean, median and standard deviations were recorded for each index. Comparisons among runs involved using one-tailed Wilcoxon rank sum testing to determine significant results. Wilcoxon tests determined whether the reduced models produced better results than the non-reduced models and whether the seeded program produced better clustering results than the original program.

To understand the clustering results, the clustering from each model with the highest RS value was selected for visualization. In the literature, clustering visualizations used simple dependency graphs as in the work of Dietrich et al. (2008) and Cassell et al. (2011). Some AOP visualization studies showed interactions among aspects and joinpoints, including Yin, Bockisch, and Aksit (2012) and Yin (2013), but none had combined clustering visualization and interaction visualization. Fabry et al. (2011) produced a zoomable interface that allowed closer inspection program elements. The idea of a zoomable visualization was combined with the clustering results of this study to

show the impact of a cluster based on potential interference.

Results were divided into two groups—one for a smaller program, AspectTetris (AT), and another for a larger program, AJHotDraw (AJHD). Each program was tested in its original form (α), and in an altered form (β) designed to increase the potential for interference. In addition, models reduced by principal component analysis were denoted with Π. Metrics were collected for the following 22 models:

- AT α M_{IP}, AT α M_{ICP}, AT α M_{CAE}
- AT β M_{IP}, AT β M_{ICP}, AT β M_{CAE}
- AT α M_{IP} Π, AT α M_{ICP}Π
- AT β M_{IP}Π, AT β M_{ICP}Π
- AJHD α M_{IP}, AJHD α M_{ICP}, AJHD α M_{CAE}
- AJHD β M_{IP}, AJHD β M_{ICP}, AJHD β M_{CAE}
- AJHD α M_{IP} Π, AJHD α M_{ICP}Π, AJHD α M_{CAE}Π
- AJHD β M_{IP}Π, AJHD β M_{ICP}Π, AJHD β M_{CAE}Π

Results showed that both the IP and ICP metrics produced more clustering improvements than the CAE metric, highlighting the fact that CAE hid too much detail to effectively describe interference problems. The IP and ICP clusterings generally showed improvements when comparing the β program with the α program. This shows that clustering successfully recognized the increased dissimilarity among clusters that resulted from introduced interference potential. Wilcoxon rank sum tests showed that PCA-reduced models generally produced better clustering results than the full versions. In addition, Wilcoxon rank sum test results showed that Dunn's index was a poor measurement tool when comparing clusterings in which the value of K fluctuates.

Looking at the clusters visually, ICP clusters produced dramatic results compared to IP clusters (even though IP clusters showed changes). This emphasized the difficulty of altering the program such that a method called many individual methods to raise the single method's IP. Results also indicated that the clustering for the AJHD program needed improvement—probably stemming from a poorly selected value for K.

These results helped to answer the research questions posed by this study. First, it was noted that IP and ICP were fine-grained metrics that describe interference potential and interference causality potential. However, the IP metric may require further study to show its true potential. Second, clustering analysis using these metrics proved to be a successful way to detect increased potential for interference problems, by generally showing positive clustering changes when introducing the seeded aspect. Third, the process proved to be somewhat scalable from a smaller program to a larger program, but clusterings were of lower quality in the larger program. The larger-scale program showed positive results, but to a lesser extent than the smaller program. Clustering metrics and inspection of clusters seemed to suggest that the optimal value for K was not found in all cases for the large program, resulting in similar clusters.

Finally, this work has opened several opportunities for future research. Numerous areas for improving the current study emerged from the study results. Other broader recommendations included improving generic metrics for interference problems, studying the effects of compilation on interference problems, viewing the IP and ICP metrics from a source-code standpoint rather than the bytecode versions presented here, creating new IP- and ICP-based metrics as ratios of the current metric value to the maximum metric in the system, and performing HCI studies on the visualization techniques used.

The software engineering discipline has many facets. This study has highlighted the need for meaningful metrics, ways of increasing the understanding of a program's structure through program analysis, and the ability to visualize the inner-workings of a system. Each of the avenues presented within this work have implications for future study. Together, these areas show the vibrant nature and countless possibilities that exist in software engineering research.

References

AJHotDraw. (2007). Retrieved February 25, 2015, 2015, from http://sourceforge.net/projects/ajhotdraw/files/ajhotdraw/AJHotDraw v.0.4/

Apel, S. (2010). How AspectJ is used: An analysis of eleven AspectJ programs. *The Journal of Object Technology, 9*(1), 117-142. doi: 10.5381/jot.2010.9.1.a2

Arthur, D., & Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 1027-1035.

The AspectJ™ Programming Guide. (2003). Retrieved September 9, 2013, from http://eclipse.org/aspectj/doc/released/progguide/index.html

Bernardi, M. L., & Di Lucca, G. A. (2010). A metric model for aspects' coupling. *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, 59-66. doi: 10.1145/1809223.1809232

Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering, SE-12*(2), 211-221. doi: 10.1109/tse.1986.6312937

Bostock, M. (2012). D3.js - Data-Driven Documents. Retrieved October 17, 2013, from http://d3js.org/

Bostock, M. (2013). Zoomable Circle Packing. Retrieved March 27, 2015, from http://bl.ocks.org/mbostock/7607535

Burrows, R., Ferrari, F. C., Garcia, A., & Taïani, F. (2010). An empirical evaluation of coupling metrics on aspect-oriented programs. *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, 53-58. doi: 10.1145/1809223.1809231

Capretz, L. F. (2003). A brief history of the object-oriented approach. *ACM SIGSOFT Software Engineering Notes, 28*(2), 6. doi: 10.1145/638750.638778

Cassell, K., Anslow, C., Groves, L., & Andreae, P. (2011). Visualizing the refactoring of classes via clustering. *Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113*, 63-72.

Ceccato, M., & Tonella, P. (2004). Measuring the effects of software aspectization. *Proceedings of the 1st Workshop on Aspect Reverse Engineering*.

Chen, X., Ye, N., & Ding, W. (2010). A formal approach to analyzing interference problems in aspect-oriented designs. In S. Qin (Ed.), *Unifying Theories of Programming* (Vol. 6445, pp. 157-171): Springer Berlin Heidelberg.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476-493. doi: 10.1109/32.295895

Czibula, G., Cojocar, G. S., & Czibula, I. G. (2009). A partitional clustering algorithm for crosscutting concerns identification. *Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems*, 111-116.

Czibula, I. G., & Şerban, G. (2006). Improving systems design using a clustering approach. *IJCSNS International Journal of Computer Science and Network Security, 6*(12), 40-49.

D'Ursi, A. C., Cavallaro, L., & Monga, M. (2007). On bytecode slicing and aspectJ interferences. *Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, 35-43. doi: 10.1145/1233833.1233839

Dahl, O.-J., & Hoare, C. A. R. (1972). Hierarchical program structures. In O. J. Dahl, E. W. Dijkstra, & C. A. R. Hoare (Eds.), *Structured programming* (pp. 175-220): Academic Press Ltd.

De Borger, W., Lagaisse, B., & Joosen, W. (2009). A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, 173-184. doi: 10.1145/1509239.1509263

Delamare, R., & Kraft, N. A. (2012). *A genetic algorithm for computing class integration test orders for aspect-oriented systems*. Paper presented at the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST).

Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G., & Duchrow, M. (2008). Cluster analysis of Java dependency graphs. *Proceedings of the 4th ACM symposium on Software visualization*, 91-94. doi: 10.1145/1409720.1409735

Disenfeld, C., & Katz, S. (2012). A closer look at aspect interference and cooperation. *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, 107-118. doi: 10.1145/2162049.2162063

Disenfeld, C., & Katz, S. (2013). Specification and verification of event detectors and responses. *Proceedings of the 12th annual international conference on Aspect-oriented software development*, 121-132. doi: 10.1145/2451436.2451452

Douence, R., Fradet, P., & Südholt, M. (2002). A framework for the detection and resolution of aspect interactions. In D. Batory, C. Consel, & W. Taha (Eds.), *Generative programming and component engineering* (Vol. 2487, pp. 173-188): Springer Berlin Heidelberg.

Evertsson, G. (2003, January 20, 2003). Tetris in AspectJ. Retrieved February 26, 2015, 2015, from http://www.guzzzt.com/files/coding/aspecttetris.pdf

Fabry, J., Kellens, A., & Ducasse, S. (2011, 22-24 June 2011). *AspectMaps: A Scalable Visualization of Join Point Shadows.* Paper presented at the Program Comprehension (ICPC), 2011 IEEE 19th International Conference on.

Figueroa, I. (2013). Towards control of aspect interference using membranes and monads. *Proceedings of the 12th Annual International Conference Companion on Aspect-Oriented Software Development*, 27-28. doi: 10.1145/2457392.2457404

Figueroa, I., Tabareau, N., & Tanter, É. (2013). Taming aspects with monads and membranes. *Proceedings of the 12th Workshop on Foundations of Aspect-Oriented Languages*, 1-6. doi: 10.1145/2451598.2451600

Halkidi, M., Batistakis, Y., & Vazirgiannis, M. (2001). On clustering validation techniques. *Journal of Intelligent Information Systems, 17*(2-3), 107-145. doi: 10.1023/A:1012801612483

Halkidi, M., Batistakis, Y., & Vazirgiannis, M. (2002). Clustering validity checking methods: part II. *SIGMOD Rec., 31*(3), 19-27. doi: 10.1145/601858.601862

Halkidi, M., Vazirgiannis, M., & Batistakis, Y. (2000). Quality Scheme Assessment in the Clustering Process. In D. Zighed, J. Komorowski, & J. Żytkow (Eds.), *Principles of Data Mining and Knowledge Discovery* (Vol. 1910, pp. 265-276): Springer Berlin Heidelberg.

Hannemann, J., & Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. *ACM SIGPLAN Notices, 37*(11), 161-173. doi: 10.1145/583854.582436

Hannousse, A., Douence, R., & Ardourel, G. (2011). Static analysis of aspect interaction and composition in component models. *ACM SIGPLAN Notices, 47*(3), 43-52. doi: 10.1145/2189751.2047871

Hilsdale, E., & Hugunin, J. (2004). Advice weaving in AspectJ. *Proceedings of the 3rd international conference on Aspect-oriented software development*, 26-35. doi: 10.1145/976270.976276

Hoare, C. A. R., He, J. (1998). *Unifying Theories of Programming*. Englewood Cliffs: Prentice-Hall.

Hussain, A., & Rahman, M. S. (2013). A new hierarchical clustering technique for restructuring software at the function level. *Proceedings of the 6th India Software Engineering Conference*, 45-54. doi: 10.1145/2442754.2442761

Jianjun, Z. (2003). Data-flow-based unit testing of aspect-oriented programs. *Proceedings of the 27th Annual International Computer Software and Applications Conference, 2003*, 188-197. doi: 10.1109/cmpsac.2003.1245340

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In M. Akşit & S. Matsuoka (Eds.), *ECOOP'97 — Object-Oriented Programming* (Vol. 1241, pp. 220-242): Springer Berlin Heidelberg.

Kofler, P. (2011). javaclass-rb. Retrieved October 17, 2014, from https://code.google.com/p/javaclass-rb/

Kumar, A., Kumar, R., & Grover, P. S. (2009). Generalized coupling measure for aspect-oriented systems. *ACM SIGSOFT Software Engineering Notes, 34*(3), 1-6. doi: 10.1145/1527202.1527209

Lanza, M., & Marinescu, R. (2006). *Object-Oriented Metrics in Practice*: Springer Berlin Heidelberg.

Lauret, J., Fabre, J.-C., & Waeselynck, H. (2011). Detecting interferences in aspect oriented programs. *Proceedings of the 13th European Workshop on Dependable Computing*, 93-98. doi: 10.1145/1978582.1978602

Lauret, J., Waeselynck, H., & Fabre, J.-C. (2012). Detection of Interferences in Aspect-Oriented Programs Using Executable Assertions. *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 165-170. doi: 10.1109/issrew.2012.34

Mantel, N. (1967). The detection of disease clustering and a generalized regression approach. *Cancer Research, 27*(2), 209-220.

Marot, A., & Wuyts, R. (2009). Detecting unanticipated aspect interferences at runtime with compositional intentions. *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, 1-5. doi: 10.1145/1562860.1562863

Marot, A., & Wuyts, R. (2010). Composing aspects with aspects. *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 157-168. doi: 10.1145/1739230.1739249

Moldovan, G. S., & Şerban, G. (2006). Aspect mining using a vector-space model based clustering approach. *Proceedings of Linking Aspect Technology and Evolution (LATE) Workshop*, 36-40.

Pan, N., & Song, E. (2012). An aspect-oriented testability framework. *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, 356-363. doi: 10.1145/2401603.2401682

Piveta, E. K., Moreira, A., Pimenta, M. S., Araújo, J., Guerreiro, P., & Price, R. T. (2012). An empirical study of aspect-oriented metrics. *Science of Computer Programming, 78*(1), 117-144. doi: 10.1016/j.scico.2012.02.003

R Core Team. (2013). *R: A language and environment for statistical computing.* from http://www.r-project.org/

Rand McFadden, R., & Mitropoulos, F. J. (2012). Aspect mining using model-based clustering. *Proceedings of IEEE Southeastcon, 2012*, 1-8. doi: 10.1109/SECon.2012.6196984

Rentsch, T. (1982). Object oriented programming. *ACM SIGPLAN Notices, 17*(9), 51-57. doi: 10.1145/947955.947961

Şerban, G., & Moldovan, G. S. (2006). A new k-means based clustering algorithm in aspect mining. *Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2006. SYNASC '06.*, 69-74. doi: 10.1109/synasc.2006.5

Shaw, M. (1989). Larger scale systems require higher-level abstractions. *ACM SIGSOFT Software Engineering Notes, 14*(3), 143-146. doi: 10.1145/75200.75222

Shepherd, D., & Pollock, L. (2005). Interfaces, aspects, and views: The discoveries of a clustering aspect miner and viewer. *Linking Aspect Technology and Evolution (LATE) Workshop*, 1-6.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices, 21*(11), 38-45. doi: 10.1145/960112.28702

Stoerzer, M., & Graf, J. (2005). Using pointcut delta analysis to support evolution of aspect-oriented software. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.*, 653-656. doi: 10.1109/icsm.2005.99

Tian, K., Cooper, K., & Zhang, K. (2010). A framework based approach for unified detection of Aspect Weaving Problems. *Proceedings of the 2010 IEEE International Conference on Information Reuse and Integration (IRI)*, 132-140. doi: 10.1109/iri.2010.5558950

Tian, K., Cooper, K., Zhang, K., & Liu, S. (2010). Towards a new understanding of advice interference. *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 180-189. doi: 10.1109/ssiri.2010.18

Tian, K., Cooper, K., Zhang, K., & Yu, H. (2009). A classification of aspect composition problems. *Proceedings of the Third International Conference on Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009.*, 101-109. doi: 10.1109/ssiri.2009.33

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*, 220-225. doi: 10.1145/2184512.2184564

Walker, R. J., Baniassad, E. L. A., & Murphy, G. C. (1999). An initial assessment of aspect-oriented programming. *Proceedings of the 21st International Conference on Software Engineering,* 120-130. doi: 10.1145/302405.302458

Wettel, R., & Lanza, M. (2008). Visually localizing design problems with disharmony maps. *Proceedings of the 4th ACM symposium on Software visualization*, 155-164. doi: 10.1145/1409720.1409745

Yin, H. (2013). A graphical tool for observing state and behavioral changes at join points. *Proceedings of the 12th Annual International Conference Companion on Aspect-Oriented Software Development*, 29-30. doi: 10.1145/2457392.2457405

Yin, H., Bockisch, C., & Aksit, M. (2012). A fine-grained debugger for aspect-oriented programming. *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, 59-70. doi: 10.1145/2162049.2162057

Yu, L., & Ramaswamy, S. (2007). Verifying design modularity, hierarchy, and interaction locality using data clustering techniques. *Proceedings of the 45th annual southeast regional conference*, 419-424. doi: 10.1145/1233341.1233417

Yu, L., & Ramaswamy, S. (2009). An empirical approach to evaluating dependency locality in hierarchically structured software systems. *Journal of Systems and Software, 82*(3), 463-472. doi: 10.1016/j.jss.2008.07.020

Zhao, J. (2004). *Measuring coupling in aspect-oriented systems.* Paper presented at the 10th International Software Metrics Symposium (METRICS), Chicago, Illinois, USA.