Supporting Feature-Level Software Maintenance

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philospohy

---

by

Meghan Revelle

2010

ABSTRACT

Software maintenance is the process of modifying a software system to fix defects, improve performance, add new functionality, or adapt the system to a new environment. A maintenance task is often initiated by a bug report or a request for new functionality. Bug reports typically describe problems with incorrect behaviors or functionalities. These behaviors or functionalities are known as features. Even in very well-designed systems, the source code that implements features is often not completely modularized. The delocalized nature of features makes maintaining them challenging. Since maintenance tasks are expressed in terms of features, the goal of this dissertation is to support software maintenance at the feature-level. We focus on two tasks in particular: feature location and impact analysis via feature coupling.

Feature location is the process of identifying the source code that implements a feature, and it is an essential first step to any maintenance task. There are many existing techniques for feature location that incorporate various types of analyses such as static, dynamic, and textual. In this dissertation, we recognize the advantages of leveraging several types of analyses and introduce a new approach to feature location based on combining dynamic analysis, textual analysis, and web mining algorithms applied to software. The use of web mining for feature location is a novel contribution, and we show that our new techniques based on web mining are significantly more effective than the current state of the art.

After using feature location to identify a feature's source code, maintenance can be completed on that feature. Impact analysis should then be performed to revalidate the system and determine which other features may have been affected by the modifications. We define three feature coupling metrics that capture the relationship between features based on structural information, textual information, and their combination. Our novel feature coupling metrics can be used for impact analysis to quantify the strength of coupling between pairs of features. We performed three empirical studies on open-source software systems to asses the feature coupling metrics and established three major results. First, there is a moderate to strong statistically significant correlation between feature coupling and faults. Second, feature coupling can be used to correctly determine about half of the other features that would be affected by a change to a given feature. Finally, we found that the metrics align with developers' opinions about pairs of features that are actually coupled.

Table of Contents

Dedicated to my husband.

ACKNOWLEDGMENTS

It would have been next to impossible to write this dissertation, or even get to the point where I could write it, without the help, guidance, and support of many individuals, both internal and external to the William and Mary community. The most important person I have to thank at W&M is my advisor, Denys Poshyvanyk. I credit him and his excellent research vision for getting me through these last two years of grad school. I also thank my committee for all their comments that helped shape and improve my work. Likewise, I am grateful to each of the members of the Semeru lab for all the contributions they made to this dissertation. I would also like to express my gratitude to the department's office staff for answering all those questions I had over the years.

I received immeasurable amounts of encouragement and moral support from my family and friends. Thank you to my parents for inspiring me not to give up when times got tough. I am indebted to my friends for always being there to help me celebrate my achievements and move past my mistakes.

Most of all, I thank my husband, Jeff. He provided the motivation and support I needed to finish. It is amazing to have a spouse who loves you so much, can answer LaTeX questions, and set up a CVS repository for you.

Financial support for this work was provided by the Air Force Office of Scientific Research, the Virginia Space Grant Consortium, and the National Science Foundation.

List of Tables

List of Figures

Supporting Feature-Level Software Maintenance

Chapter 1

Introduction

Software maintenance is the process of modifying a software system after its initial development and deployment [209]. Software systems undergo changes for a variety of reasons: to fix problems, to improve performance, to add new functionalities, or to be adapted to a new environment. The software maintenance process has three main steps [17, 16]. First, the programmer must understand the existing software, at least partially. This crucial step can take 50%-60% of the total time required for maintenance [51, 115, 232]. Second, once adequate comprehension is achieved, the programmer can modify the software. Finally, the programmer must revalidate the newly modified software to ensure proper functionality and performance.

The software maintenance process is often triggered by a bug report or a feature request. Bug reports typically describe problems related to incorrect system behaviors or functionalities. These program behaviors or functionalities are known as *features*[1]. In the literature, a feature is defined as "a requirement of a program that a user can exercise and which produces an observable behavior" [5]. For instance, an example of a feature from a web browser is the ability to save a bookmark for a web page. As another example, features of a word processor include the abilities to spell-check and print a document.

Since many maintenance tasks are initiated by bug reports, and most bug reports are expressed in terms of features, it is the goal of this work to support software maintenance

[1]Features are also sometimes referred to as concepts or concerns [175, 176]. The definitions of "concept" and "concern" are broader than the definition of "feature" because they cover behaviors of a software system that users cannot invoke or observe. This dissertation focuses on features.

tasks at the feature-level and to promote features to first-class entities [216] within at least one phase of the software life cycle. We focus on two software maintenance activities in particular: feature location and impact analysis.

Feature location, also referred to as *concept location*, is the process of identifying the source code that implements a feature [5, 12]. Before maintainers can change a system, they must explore its source code in order to locate and understand the code that is relevant to the feature undergoing modification. Thus, feature location corresponds to the first step of the software maintenance process. Locating a feature's source code is a challenging task, especially in large systems with hundreds of classes, thousands of methods, and millions of lines of code. Compounding the problem is the fact that features' implementations are often not encapsulated in a single module [61, 111, 174, 213]. Even in well-designed systems, it is inevitable that some features will have to be implemented in multiple modules.

Because features are scattered throughout the modules of a system, they are hard to locate and maintain. Feature location techniques seek to help maintainers more effectively and efficiently identify a feature's source code. Most existing feature location techniques locate features using textual, dynamic, or static analyses. Textual approaches leverage the semantic information embedded in source code comments and identifier names [45, 85, 103, 142, 157, 201]. However, if the system has poor identifier names, textual feature location techniques may not perform well. Dynamic approaches identify a feature's relevant source code by analyzing execution traces [5, 65, 66, 77, 229]. These dynamic techniques are prone to 1) being noisy because of the difficulty of only invoking the feature of interest at runtime and 2) incomplete because all of a feature's relevant source code may not be executed in a trace. Static feature location techniques generally require more user input and involve a programmer exploring the structural dependencies of code known to implement a feature to find additional relevant code [39, 176, 214].

Researchers have recognized that textual, dynamic, and static feature location techniques have their limitations, but by combining them, their weaknesses can be mitigated [5, 65, 66, 77, 229]. This work also introduces new feature location techniques that are based on combining several types of analyses such as textual, dynamic, and static. In addition, this work introduces the use of web mining algorithms for feature location and shows

that these new approaches are significantly more effective than existing techniques.

Once maintainers locate and understand a feature's source code, they can make appropriate changes to the feature to satisfy the maintenance task. These actions cover the first two steps of the software maintenance process. The third step is to revalidate the software, and this step can be achieved by performing impact analysis. *Impact analysis* is the process of determining the effects of a change to a software system [18, 40, 189]. One way of performing impact analysis is to use coupling metrics [29]. For instance, if class A is modified and is also tightly coupled to class B, then B is likely to be affected by changes to A.

Existing coupling metrics are defined for classes. However, features transcend the boundaries of classes, so these existing metrics cannot be applied to them. To more effectively support software maintenance of features, this work introduces metrics that are specifically designed to capture the coupling among features. Maintainers can use these metrics to determine whether other features might be affected by the changes they make. The new metrics capture the coupling among features by relying on structural and textual sources of information in source code.

This dissertation takes a novel view of supporting software maintenance by focusing on features. This work concentrates on the software maintenance tasks of feature location and feature-level impact analysis. Thus, this dissertation provides a comprehensive approach to supporting feature-level software maintenance tasks.

1.1 Research Goals and Contributions

Since the implementations of features are not always modularized, they are difficult to locate. Also, determining the relationships between un-modularized features is challenging. This work aims to expressly support two maintenance tasks in terms of features: feature location and feature impact analysis via coupling. Both of these tasks are achieved by combining information from different sources, a process known as *data fusion* [112]. The principle behind data fusion is that combining information from different sources yields better results than if the data sources were used individually. This idea has been successfully

applied to feature location [62, 76, 102, 130, 160, 244] as well as other areas of software engineering research [60, 79, 223]. The sources of data that can be analyzed from software are structural dependencies among program elements, execution information derived dynamically at runtime, and textual information embedded in the identifiers and comments found in source code. This work proposes to combine these sources of information to support both feature location and feature coupling.

Specifically, this dissertation makes the following research contributions:

1. **A survey of feature location research**.

   We have conducted a comprehensive survey of existing feature location research and classified the literature within a taxonomy that has nine dimensions. The taxonomy captures key facets of typical feature location techniques and can be useful to both software engineering researchers and practitioners. Researchers can use this survey to identify related work as well as opportunities for future research. Practitioners can use this overview to determine which feature location approach is most suited to their needs.

2. **An exploration of the use of several types of analyses for feature location**.

   We carried out an exploratory study of ten feature location techniques that use various combinations of textual, dynamic, and static analyses. A new way of applying textual analysis is introduced by which queries are automatically composed of the identifiers of a method known to be relevant to a feature. Our results show that this new type of query is just as effective as a query formulated by a human. We also provide insight into situations when certain feature location approaches are successful and unsuccessful. The results and observations of this exploratory study were used to guide the direction of the feature location research presented in this dissertation.

3. **Development and evaluation of new feature location technique based on web mining**.

   We created a data fusion model for feature location which defines new feature location techniques based on combining information from textual, dynamic, and web mining analyses applied to software. A novel contribution of the proposed model is the use of

web mining algorithms to analyze execution information during feature location. The results of an extensive evaluation indicate that the new feature location techniques based on web mining improve the effectiveness of existing approaches by as much as 62%.

4. **Development and evaluation of feature coupling metrics**.

   We have defined new feature coupling metrics based on structural and textual source code information and extended the unified framework for coupling measurement to include these new metrics. We also conducted three extensive case studies to evaluate these new metrics. The first study examined the relationship between feature coupling and fault-proneness, the second assessed feature coupling in the context of impact analysis, and the third study surveyed developers to determine if the metrics align with what they consider to be coupled features. All three studies provide evidence that feature coupling metrics are indeed useful new measures that capture coupling at a higher level of abstraction than classes and can be useful for finding bugs, guiding testing efforts, and assessing change impact.

5. **Tool support for feature location and feature coupling**.

   We have developed a tool called FLAT^3 that integrates textual and dynamic feature location techniques along with feature annotation capabilities, a useful visualization technique, and the ability to compute the proposed feature coupling metrics. FLAT^3 provides a complete suite of tools that allows developers to quickly and easily locate the code that implements a feature, save these annotations for future use, and compute feature coupling based on the saved annotations.

1.2 Scope of this Dissertation

Features, have been extensively studied in the literature. To clarify the scope of this dissertation, we discuss some of the research related to features and how our work differs. In this section, we cover other programming paradigms that have been introduced to work with features and research on feature analysis.

Feature-oriented programming (FOP) [10, 11] focuses on creating software product lines, which are families of software systems in which each program is composed of a unique set of features. In FOP, programs are layered, and each layer adds a new feature, thus features are modularized. FOP is a programming paradigm for synthesizing software, and there are also approaches for refactoring and remodularizing object-oriented systems into the FOP paradigm [131, 153]. In our work, we do not introduce any new paradigms but seek to support the maintenance of features within the object-oriented paradigm where features cannot always be modularized.

Aspect-oriented programming (AOP) [111] is a paradigm that seeks to solve the problem of un-modularized features. In AOP, a new language construct called an aspect is created that modularizes a feature's implementation. Then a specialized compiler, called a weaver, follows instructions, called advice, on where to inject the aspect into the code base. AOP works well for a few types of features, such as logging, which have implementations that can be easily injected automatically by the weaver. However, AOP cannot entirely solve the problem of un-modularized features, and our work helps support maintainers who have to deal with the lack of localization.

Besides programming paradigms meant to cope with features, there is a wealth of existing research on feature analysis, in which features are considered first-class entities of a software system. These approaches have focused on how programmers develop features [94], a feature-centric environment for source code browsing [188], identifying canonical sets of features [120, 121, 119], reverse engineering [91], and identifying and refactoring features that need evolution [149]. Our work focuses on locating features' implementations and on determining the relationships between features using coupling.

1.3 Dissertation Organization

Chapter 2 presents our survey of feature location literature. Eighty-eight research, tool, case, industrial, and user study articles from 30 software engineering venues have been reviewed and classified within a taxonomy in order to organize and structure existing work in the field of feature location.

Chapter 3 presents the results of an exploratory study of feature location techniques based on combining textual, dynamic, and static analyses. The goal of the study was to examine how well these techniques locate multiple methods that are relevant to a feature, whereas most previous studies focus on how well feature location techniques find a single relevant method. In addition, different parameters to the analyses used are explored. The results of a user study comparing the various techniques with different parameters are presented.

Chapter 4 introduces a data fusion model for feature location. The model is based on combining textual analysis, dynamic analysis, and web mining. Web mining is a branch of data mining that extracts useful information from the structure of the World Wide Web. We employ web mining algorithms to extract useful information from a program's call graph. The extracted information is used to effectively filter out false positives from the results of a feature location technique based on combining textual and dynamic analyses. The results of two case studies on open source systems are presented.

Chapter 5 defines three feature coupling metrics. One metric is based on structural information, one is based on textual information, and the final metric is based on a combination of structural and textual information. The unified framework for coupling measurement [25] is extended to include these new metrics. The chapter also reports the results of three separate evaluations aimed at answering the question, "Are feature coupling metrics useful?"

Chapter 6 presents a tool called FLAT^3, the **F**eature **L**ocation and **T**extual **T**racing **T**ool, that implements the ideas proposed in this dissertation. FLAT^3 supports several types of feature location. These techniques can be used to find features' implementations, and then the located code can be associated with features. These associations can be used to automatically compute our feature coupling metrics and to visualize the distribution of a feature throughout a system's classes.

1.4 Bibliographical Notes

Portions of this dissertation have been previously published or have been submitted for publication and are under review at the time of this writing. Chapter 2 is based on a submission to an international journal. Material from Chapter 3 was published and presented at the 17th International Conference on Program Comprehension (ICPC 2009) [173]. Portions of the work presented in Chapter 4 have been accepted for publication at the 18th International Conference on Program Comprehension (ICPC 2010) and also submitted to an international journal. The ideas and results in Chapter 5 are currently under review for publication at an international journal. Finally, parts of Chapter 6 appear in a formal research demonstration accepted for publication at the 32nd International Conference on Software Engineering (ICSE 2010) [198].

Chapter 2

Survey of Feature Location Research

In software systems, a feature represents a functionality that is defined by requirements and accessible to developers and users. Software maintenance and evolution involves adding new features to programs, improving existing functionalities, and removing bugs. Identifying the parts of the source code that correspond to a specific functionality is known as *feature* (or *concept*) *location* [12, 168]. It is one of the most frequent maintenance activities undertaken by developers because it is part of the incremental change process [167]. During the incremental change process, programmers use feature location to find where in the code the first change to complete a task needs to be made. The full extent of the change is then handled by impact analysis, which starts with the source code found by feature location and finds all code affected by the change. Methodologically, the two activities of feature location and impact analysis are different and are treated separately in the literature.

Feature location is one of the most important and common activities performed by programmers during software maintenance and evolution. No maintenance activity can be completed without first locating the code that is relevant to the task at hand, making feature location essential to software maintenance. For example, Alice is a new developer on a software project, and her manager has given her the task of fixing a bug that has been reported. Since Alice is new, she is unfamiliar with the large code base of the software system and does not know where to begin. Lacking sufficient documentation on the system

and the ability to ask the code's original authors for help, the only option Alice sees is to manually search for the code relevant to her task.

Alice's situation is one faced by many programmers needing to understand and modify an unfamiliar codebase. However, a manual search of a large amount of source code, even with the help of tools such as pattern matchers or an integrated development environment, can be frustrating and time-consuming. Recognizing this problem, software engineering researchers have developed numerous feature location techniques to aid programmers in Alice's position. The various techniques that have been introduced are all unique in terms of their input requirements, how they locate a feature's implementation, and how they present their results. Thus, even the task of choosing a suitable feature location technique can be challenging.

The existence of such a large body of feature location research calls for a comprehensive overview. Since there currently is no broad summary of the field of feature location, this chapter provides a survey and operational taxonomy of this pertinent research area. The survey includes research articles that introduce new feature location approaches; case, industrial, and user studies; and tools that can be used in support of feature location. The articles are characterized within a taxonomy that has nine dimensions, and each dimension has a set of attributes associated with it. The dimensions and attributes of the taxonomy capture key facets of typical feature location techniques and can be useful to both software engineering researchers and practitioners [140]. Researchers can use this survey to identify what has been done in the area of feature location and what needs to be done; that is, they can use it to find related work as well as opportunities for future research. Practitioners can use this overview to determine which approach is most suited to their needs.

This survey encompasses 88 articles (51 research articles and 37 tool and case study papers) from 30 venues published before October 2009. The research articles were selected because they either state feature/concept location as their goal or present a technique that is essentially equivalent to feature location. The tool papers include tools developed specifically for feature location as well as program exploration tools that support feature location. The case study articles include industrial and user studies as well as studies that compare existing approaches.

There are several research areas that are closely related to feature location such as traceability link recovery, impact analysis, and aspect mining. Traceability link recovery seeks to connect documentation with source code, while feature location is more concerned with identifying source code associated with functionalities, not specific sections of a document. Impact analysis is the step in the incremental change process performed after feature location with the purpose of expanding on feature location's results, especially after a change is made to the source code. Feature location focuses on finding the starting point for that change. The main goal of aspect mining is to identify cross-cutting concerns and determine the source code that should be refactored into aspects, meaning the aspects themselves are not known a priori. By contrast, in the contexts in which feature location is used, the features are already known and only the code that implements them is unknown. Therefore, articles and research from these related fields are not included here as they are beyond the scope of this survey.

The work presented in this chapter has two main contributions. The first is a comprehensive survey of feature location techniques, relevant case studies, and tools. The second is a taxonomy derived from those techniques. Appendix A lists all of the surveyed articles classified within the taxonomy. Section 2.1 introduces the dimensions of the taxonomy, and Section 2.2 provides brief descriptions of the surveyed research articles. Section 2.3 overviews the tools and studies, and Section 2.4 discusses open issues in feature location. Section 2.5 concludes.

2.1 Dimensions of the Survey

The goal of this survey is to provide researchers and practitioners with an organized overview of existing feature location research. From a thorough inspection of the literature, a number of key dimensions emerged. These dimensions objectively describe the different techniques and give structure to the surveyed literature. The dimensions are as follows.

- The type of the article: Is it a research article, case study, or tool paper?
- The type of analysis: What analyses are used to support feature location?

- The sources of information: What sources of information are used for feature location?
- The granularity: What is the granularity of the located program elements?
- Programming language support: To what languages has the technique been applied?
- The presentation of the results: How are the located program elements presented to the user?
- The evaluation of the approach: How is the technique assessed?
- Comparison to other feature location techniques: To what other approaches is the new one compared, if any?
- The systems used for evaluation: To what software systems has the technique been applied?

The order in which these dimensions are presented does not imply any explicit priority or importance.

Each dimension has a number of distinct attributes associated with it. For a given dimension, a feature location technique may be associated with multiple attributes. These dimensions and their attributes were derived by examining an initial set of articles of interest. They were then refined and generalized to succinctly characterize the properties that 1) make feature location techniques unique and 2) can be used to evaluate and compare them. The goal of the taxonomy is to allow researchers and practitioners to easily locate the feature location techniques that are most suited to their needs. The dimensions and their associated attributes are listed in Table A.1 in Appendix A along with their abbreviations (given in parentheses) that are used in the taxonomy of the surveyed articles. These dimensions and attributes are discussed in the remainder of this section.

2.1.1 Type of Article

This survey encompasses three types of feature location articles: research articles, tool papers, and case studies. The research articles introduce new feature location techniques. The tool papers describe applications that perform feature location and program exploration tools that, while not necessarily designed for feature location, support the search for a feature's implementation. The case study articles include direct comparisons of several feature location techniques, industrial case studies, and user studies.

2.1.2 Type of Analysis

A primary distinguishing factor of feature location techniques is the type, or types, of analysis they employ to identify the code that pertains to a feature. The most common analyses include dynamic, static, and textual. While these are not the only analyses possible, they are the ones used by the vast majority of feature location techniques, and some approaches even leverage more than one of these analyses. In Section 2.2, descriptions of all the surveyed articles are given, and the section is organized by the types of analyses used.

Dynamic analysis refers to examining a software system's execution, and it is often used for feature location when features can be invoked and observed during runtime. Feature location using dynamic analysis generally relies on a post-mortem analysis of an execution trace. Typically, one or more feature-specific scenarios are developed that invoke only the desired feature. Then, the scenarios are run and execution traces are collected, recording information about the code that was invoked. These traces are captured either by instrumenting the system or through profiling. Once the traces are obtained, feature location can be performed in many ways. The traces can be compared to other traces in which the feature was not invoked to find code only invoked in the feature-specific traces [76, 229]. Alternatively, the frequency of execution of portions of code can be analyzed to locate a feature's implementation [5, 77, 190]. Using dynamic analysis for feature location is a popular choice since most features can be mapped to execution scenarios. However, there are some limitations associated with dynamic analysis. The collection of traces can impose significant overhead on a system's execution. Additionally, the scenarios used to collect traces may not invoke all of the code that is relevant to the feature, meaning some of the feature's implementation may not be located [233]. Conversely, it may be difficult to formulate a scenario that invokes only the desired feature, causing irrelevant code to be executed [68, 76, 233]. Dynamic feature location techniques are discussed in Section 2.2.1.

Static analysis examines structural information such as control and data flow dependencies. In manual feature location, developers may follow program dependencies in a section of code they deem to be relevant in order to find additional useful code, and this

idea is used in some approaches to static feature location [39]. Other techniques analyze the topology of the structural information to point programmers to potentially relevant code [176]. While using static analysis for feature location is very close to what a human searching for code may do, it often overestimates what is pertinent to a feature and is prone to giving many false positive results. Static approaches to feature location are summarized in Section 2.2.2.

Textual approaches to feature location analyze the words used in source code. The idea is that identifiers and comments encode domain knowledge, and a feature may be implemented using a similar set of words throughout a software system, making it possible to find a feature's relevant code textually. Textual analysis is performed using three main techniques: pattern matching, information retrieval (IR) and natural language processing (NLP). Pattern matching usually involves a textual search of source code using a utility such as grep[1]. Information retrieval techniques, such as Latent Semantic Indexing (LSI) and Vector Space Model (VSM), are statistical methods used to find a feature's relevant code by looking for identifiers and comments that are similar to a query provided by a user. NLP approaches can also use a query, but they analyze the parts of speech of the words used in source code. Pattern matching is relatively robust but not very precise because of the vocabulary problem [81]; the chances of a programmer choosing query terms that match the vocabulary of unfamiliar source code are very low. On the other hand, NLP is more precise than pattern matching but much more expensive. Information retrieval lies between the two. No matter the type of textual analysis used, the quality of feature location is heavily tied to the quality of the source code naming conventions and/or the user-issued query. Textual feature location techniques are reviewed in Section 2.2.3.

Feature location is not limited to just dynamic, static, or textual analysis. Many techniques draw on multiple analyses to find a feature's implementation, and some do not use any of these types of analyses. Existing approaches that combine analyses do so with the goal of using one type of analysis to compensate for the limitations of another, thus achieving better results than standalone techniques. The unique ways in which multiple types of analyses are combined for feature location are described in Sections 2.2.4 through 2.2.6.

[1] `http://www.gnu.org/software/grep/`

Other approaches do not rely on dynamic, static, or textual analysis. For instance, two feature location techniques rely on historical analysis by mining repositories in order to identify lines of code [38] or artifacts related a feature [218]. Another technique examines the code visible to a programmer during a maintenance task and tries to infer what was important [180]. These exceptions are explained in Section 2.2.8.

2.1.3 Sources of Information

Related to the types of analyses used by a feature location technique are the sources of information used by that approach. In the same way that a technique can make use of multiple types of analyses, one or more sources of information can also be used for feature location. The analysis must be performed on some artifact(s) related to the software system such as source code, documentation, execution traces, and dependence graphs. Typically, the sources of information used match the type of analysis employed. Dynamic analysis uses execution traces captured when a feature is executed. Different representations of source code, such as a call graph, can be used by static analysis. Source code and documentation can be leveraged in textual analysis to find code that is relevant to a feature. For the exceptional cases, the two that use historical analysis mines version control systems, issue trackers, and communication archives, and the other exception uses a transcript of a program investigation listing the code visible to a programmer and how it was accessed.

2.1.4 Granularity

The purpose of feature location is to find the source code that implements a specific feature. Existing feature location techniques identify code at different granularities: classes, methods, basic blocks, lines, decisions, or uses of variables. In this survey, the phrase *program elements* refers to portions of code at any of these levels of granularity. The more fine-grained the program elements located by a technique, the more specific and expressive the feature location technique is. For instance, all the basic blocks or variables may be relevant, but when classes are located, not all the methods in the class may pertain to the feature. Some approaches may be applicable to multiple levels of granularity, but only

those program elements that are actually shown to be supported in an article are reported in this survey.

2.1.5 Programming Language Support

The programming language in which a software system is written can play a factor in the types of feature location techniques that can be applied to it. Textual and historical analyses are programming language agnostic at the file level, but require parsers to be applied to methods or other levels of granularity. Static and dynamic analyses can be limited due to tool support for a given language. In this survey, all programming languages on which a technique has been applied are reported. The majority of existing feature location approaches have been exercised on Java or C/C++ systems since ample tool support is available for these languages. Other programming languages that have been supported include FORTRAN and COBOL. Knowing the languages under which an approach works can help researchers and practitioners select an appropriate technique, though the fact that an approach has not been used on a certain programming language does not imply that it is not applicable to systems implemented in that language.

2.1.6 Presentation of the Results

Once a feature location technique identifies candidate program elements for a feature, those results must be presented to the programmer. Existing feature location approaches have different ways of reporting their findings. One option is to present a list of candidate program elements ranked by their relevance to the feature [5, 77, 130, 142, 176]. Generally, the programmer only examines the top-ranked elements on the list, or only the most relevant program elements are reported to the programmer. Another way in which feature location results are presented is as an unordered set of program elements [62, 76, 229]. A set of elements is identified as being relevant to a feature, but no notion of their degree of relevance is given. Another alternative form of presentation is to visualize the software system and highlight the relevant program elements [22, 222, 235]. Finally, some feature location techniques do not automatically identify relevant program elements but describe a process a programmer can follow to manually search for a feature's implementation [39].

Since different feature location techniques present their results in different ways, comparing approaches that use different reporting styles can be challenging.

2.1.7 Evaluation

The way in which a feature location technique is evaluated provides researchers and practitioners with useful information on the approach's quality, robustness, and practical applicability. Evaluating a feature location technique is difficult because defining the program elements that are relevant to a feature is subjective. Despite this difficulty, researchers have devised a number of ways to assess feature location techniques. The most simplistic evaluations are preliminary in nature and involve a small, toy system or anecdotal evidence that the approach works. More advanced evaluations adopt a benchmark that designates the program elements that are related to a feature. Common benchmarks include documentation, patches submitted to an issue tracker, and program elements modified to fix a bug with a particular feature. Patches or bugs provide documented, reproducible gold sets of a feature's program elements. These benchmarks are generated by developers who are familiar with the software, so they carry more weight than anecdotal evidence. However, there is no guarantee that these benchmarks are 100% correct and complete. Patches and bugs may only pertain to a small portion of a feature and not touch all of its program elements. Another way to evaluate a feature location approach is to have system experts or even non-experts assess its results, which is an evaluation method often used by IR-based search engines. When multiple experts or non-experts are used, the intersection of their results can be used to create a benchmark. However, the agreement among programmers as to what program elements are relevant to a feature can be low [183].

2.1.8 Comparison to Other Feature Location Techniques

When new feature location techniques are introduced, they should be directly compared with existing approaches in order to demonstrate their (expected) superior performance. Articles that include comparisons of feature location techniques are very useful to researchers and practitioners because they highlight the advantages and limitations of the compared approaches in certain situations. Feature location techniques that appear fre-

quently in comparisons are Abstract System Dependence Graphs (ASDG) [39], Dynamic Feature Traces (DFT) [77], Formal Concept Analysis-based feature location (FCA) [76], Latent Semantic Indexing-based feature location (LSI) [142], Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) [160], software reconnaissance [229], and Scenario-based Probabilistic Ranking (SPR) [5]. UNIX grep is also another popular point of comparison because programmers often use it to textually search for relevant source code.

2.1.9 Systems used for Evaluation

A wide variety of software systems have been studied in feature location research, and the size and type of systems used in a case study reflect, to a degree, the applicability of a technique. By reviewing the software systems that have previously been used for feature location, some *de facto* benchmarks emerge. Some of the more popular systems are web browsers like Mozilla[2] , Firefox[3], Mosaic, and Chimera[4]. Other systems that have been investigated frequently are Eclipse[5], jEdit[6], and JHotDraw[7]. For some of these systems, there are a few features that are repeatedly used, but overall, no gold standard of features and their associated program elements has emerged. An abundance of other software systems have been studied. The systems on which a feature location technique has been applied are listed in the taxonomy. Having a comprehensive list of the software systems studied for feature location allows researchers to identify good candidates for systems to use in their own evaluations. Also, it lets practitioners recognize approaches that may be successfully applied to their own software system if the program they wish to apply a feature location technique to is similar to a system on which the approach has already been used.

[2] `http://www.mozilla.org/`
[3] `http://www.mozilla.org/firefox`
[4] `http://www.chimera.org/`
[5] `http://www.eclipse.org/`
[6] `http://www.jedit.org/`
[7] `http://www.jhotdraw.org/`

2.2 Survey of Feature Location Techniques

This section summarizes the 88 research, tool, and case study articles reviewed for this survey. An initial subset of articles of interest were selected, then additional relevant articles were found by following references, visiting authors' websites, and using online search tools. The articles were published in 30 different venues. Figure 2.1 shows the distribution of articles across venues, and Table 2.1 lists the abbreviations and names of the venues. The height of the bars represents the number of feature location articles published. Venues at which only one surveyed paper was published are grouped together in the "Other" bar. Black bars represent journals, and gray bars denote conferences and workshops.

When summarizing the feature location techniques and multiple articles describe a given approach (such as conference and journal versions), both are cited but the summary primarily pertains to the journal version. The articles are classified by the types of analysis used for feature location, and other dimensions of the taxonomy are mentioned as appropriate. The types of analyzes employed is the most distinguishing characteristic of feature location approaches, so it is a logical choice for the organization of this survey. In the subsections below, the surveyed articles are categorized by their use of one or more types of analyses: dynamic; static; textual; dynamic and static; dynamic and textual; static and textual; dynamic, static, and textual; and other. Table A.2 (located in Appendix A) presents the articles and their classifications within the dimensions of the taxonomy.

2.2.1 Dynamic Feature Location

Dynamic feature location relies on collecting information from a system during runtime. Dynamic analysis has a rich history in the area program comprehension [53], and feature location is just one subfield in which it is used. A number of dynamic approaches exist that deal with feature interactions [69, 192, 194], feature evolution [93], hidden dependencies among features [80], as well as identifying a canonical set of features for a given software system [120]. These techniques are beyond the scope of this survey which focuses only of approaches that seek to identify candidate program elements that implement a feature.

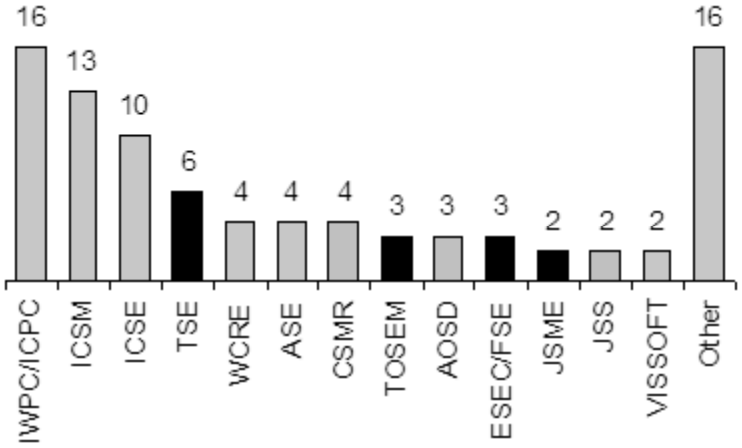


Figure 2.1: Distribution of the surveyed articles across publication venues. Black bars represent journals, and gray bars denote conferences.

Table 2.1: Venues which have published the articles included in this survey.

Acronym	Description
JSME	Journal on Software Maintenance and Evolution
JSS	Journal on Systems and Software
TOSEM	Transactions on Software Engineering
TSE	Transactions on Software Engineering and Methodology
AOSD	Aspect-Oriented Software Development
ASE	International Conference on Automated Software Engineering
CSMR	European Conference on Software Maintenance and Reengineering
ESEC/FSE	European Software Engineering Conference/Symposium on the Foundations of Software Engineering
ICSE	International Conference on Software Engineering
ICSM	International Conference on Software Maintenance
IWPC/ICPC	International Workshop/Conference on Program Comprehension
VISSOFT	International Workshop on Visualizing Software for Understanding and Analysis
WCRE	Working Conference on Reverse Engineering

This subsection summarizes articles that achieve this goal using dynamic analysis.

Software reconnaissance [228, 229] is one of the earliest feature location techniques, and it relies solely on dynamic information. Two sets of scenarios or test cases are defined, scenarios that activate a feature and scenarios that do not, and then execution traces of all the scenarios are collected. For example in a word processor, if the feature to be located is spell checking, feature-specific scenarios would activate the spell checker and the other scenarios would not. Feature location is then performed by analyzing the two sets of traces and identifying the program elements (methods) that only appear in the traces that invoked the feature. This idea of comparing traces from scenarios that do and do not invoke a feature has been heavily used and extended by other researchers in the field.

One extension to the software reconnaissance approach is Dynamic Feature Traces (DFT) [77]. First, scenarios/test cases are grouped by feature, and then execution traces are collected. Next, all pairs of method callers and callees are identified from a trace, and each method is assigned a rank for the feature. The rank is based on the average of three heuristics: multiplicity, specialization, and depth. Multiplicity is the percentage of a feature's tests that exercise a method compared to the percentage of methods in each non-feature's set of tests. Specialization is the degree to which a method was only executed by a feature and no others. Depth measures how directly a set of tests exhibits a feature compared to the other test sets. Since DFT is a refinement of software reconnaissance, the two techniques were compared head-to-head on three Java systems, finding DFTs to be more useful because developers using them are more likely to discover a feature's relevant methods.

Like software reconnaissance and DFT, Scenario-based Probabilistic Ranking (SPR) [4, 5] relies only on dynamic analysis to identify a feature's relevant program elements (methods). The idea behind SPR is to assign a probability that an event in an execution trace is associated with a feature and then rank all events. In SPR, like in software reconnaissance, two sets of scenarios are defined, scenarios that do and do not exercise a feature, and method-level execution traces are collected for each scenario. The traces are partitioned in to intervals. Intervals correspond to a sub-sequence of contiguous events (method calls) from the traces, where I is an interval from a relevant scenario, and I' is

an interval from an irrelevant scenario. Events are classified as relevant to a feature or not by determining if their frequency in interval I is greater than their frequency in interval I'. For any interval, an event's frequency is computed as the ratio of the number of times the event appears in an interval over the total number of events in the interval. Essentially, determining whether an event is relevant to a feature or not is a statistical hypothesis test. The null hypothesis is that an event's frequency in the two types of intervals is the same. A threshold, Θ, is chosen, and if an event is classified as relevant to a feature more than Θ times, the null hypothesis is rejected with a confidence level α. Events are also ranked by their relevance to a feature using a relevance index score that is computed from the number of times an event appears in relevant intervals versus the number of times it appears in irrelevant intervals. SPR has been applied to a number of systems including Mozilla, Firefox, Chimera, JHotDraw, and XFig[8]. Case studies have compared SPR directly to feature location using grep, information retrieval [142], and formal concept analysis [76]. Unlike the other two approaches in the comparison, SPR ranks its results, thus it is successful at reducing the amount of data a programmer needs to inspect to find relevant program elements.

Similar to the way SPR uses a threshold to classify relevant events from execution traces, so does the approach introduced by Safyallah and Sartipi [190]. They apply a sequential pattern mining technique to execution traces to locate the methods that implement a feature. An execution pattern is a continuous portion of a trace that appears in at least a given number (called *MinSupport*) of a feature's traces. There are strategies for identifying execution patterns for a single feature or a group of features based on setting the *MinSupport* threshold. In a case study on XFig, not only was code for the invoked features located, but execution patterns for less visible features such as mouse pointer handling and canvas view updating were identified.

While the dynamic approaches thus far have focused on locating a feature's methods or classes, Wong et al. [234] use execution slices to locate features at either the level of basic blocks, decisions, or uses of variables. A small set of carefully selected test cases that invoke the desired feature are run along with another set that does not execute the feature. From

[8] `http://www.xfig.org/`

these sets, they define heuristics for finding code that is unique to a feature or common to several features. Code that is in the union of the invoking sets but not in the union of the non-invoking sets is unique to the feature, while code in the intersection of the sets invoked by two separate features is common to those features. An evaluation of this execution slice technique was performed on five features of SHARPE [191], a stochastic model analyzer, in which system experts verified the results. Tool support for this approach was built into χVue [2], in which a system is instrumented at compile time, traces can be collected and assigned to be in the (non)invoking set, and source code related to a feature is highlighted in a development environment.

The previous approaches are susceptible to imprecision when applied to multi-threaded and distributed systems. To overcome this problem, Edwards et al. [65] developed a dynamic approach for feature location in distributed systems. The technique is based on causal relationships among events (messages) and assumes a developer can identify the first and last events associated with a feature. An interval is defined as all events that causally follow or precede a feature's starting and ending events, respectively. Program elements are assigned a component relevance index, which is the proportion of executions of that element during a feature's interval. This score can be used to rank messages, and two case studies showed that the approach ranks relevant methods highly.

A main shortcoming of dynamic analysis is the overhead it imposes on a system's execution. In distributed and time-sensitive systems, the use of dynamic analysis can be prohibitive. Edwards et al. [66] report on their experiences using dynamic analysis to perform feature location in time-sensitive systems. Instrumenting a software system in order to collect execution traces of the program elements that are invoked affects the system's runtime performance. Edwards et al. developed a minimally intrusive instrumentation technique called *minist* that reduced the number of instrumentation points while still keeping test code coverage high. For an initial evaluation, Apache's httpd[9] and several large in-house programs were used, and *minist* was compared to uninstrumented executions as well as several other tools for collecting traces. The minist approach increased execution time by only 1% on httpd, while the other tools caused increases of 7% to over 2,000%.

[9] `http://httpd.apache.org/`

2.2.2 Static Feature Location

In contrast to dynamic feature location, static feature location does not involve executing a software system. Instead, source code is statically analyzed and its dependencies and structure are explored manually or automatically. Feature location techniques leverage several types of dependencies such as control and data. The structure of a software system's dependencies can also be exploited to locate candidate program elements. Generally, static analysis can be performed to construct a dependence graph, but the actual search requires some initial starting set of relevant program elements, which means this type of analysis involves some degree of human input.

Abstract System Dependence Graphs (ASDG) [39] were introduced by Chen and Rajlich as an aid to programmers who need to find the code related to a maintenance task. It is a feature location technique based on statically-built program dependence graphs. Nodes in an ASDG correspond to either methods or global data, and edges denote either control or data dependences between nodes. Using ASDG is a manual technique whereby programmers choose a starting point (e.g., a known relevant method or *main* by default) and then search the graph using a depth-first or breadth-first strategy until all relevant program elements are found. ASDGs have been used in a case study involving the Mosaic web browser, allowing partial comprehension of the system.

Feature location with landmarks and barriers is a more automated approach than ASDGs. Developed by Walkinshaw et al. [222], it is a static feature location technique based on slicing a call graph. The first step of the approach is to identify landmark and barrier methods in a static call graph, where a landmark is a method that contributes to a feature and barriers are irrelevant methods. Direct paths between landmark nodes, known as hammock graphs, are found, and additional dependencies are obtained via backward slicing. Barriers and their dependencies are removed from the call graph to prevent exploration of irrelevant methods. The output of this approach is a pruned call graph. The technique was evaluated on NanoXML[10], Freemind[11], and JHotDraw, finding that the landmark and barrier technique substantially reduces the size of the call graph that a programmer has to

[10] `http://devkix.com/nanoxml.php`
[11] `http://freemind.sourceforge.net/`

investigate.

Another approach that is more automated than ASDGs is topology analysis of the structural dependencies [175, 176] in a software system. The main thesis of the approach is that by analyzing a program's topology (method calls and field accesses), programmers can be guided towards relevant sections of code. The algorithm begins with an initial set I of program elements identified as relevant by the programmer. The algorithm then examines the structural dependencies of the elements in I and the rest of the system to produce a suggestion set S. Both sets are actually fuzzy sets, and each element in S is assigned a value signifying its relevance. The value is based on two heuristics: *specificity* and *reinforcement*. Specificity refers to how specific or unique a structural dependency is. If program element a is in I, and its only dependency is with b, then b would be ranked highly in S. The other heuristic, reinforcement, ranks program elements highly that appear to be odd ones out. For example, if element x from I has five dependencies, and four of them are already in I, then the fifth would be ranked highly in S. Tool support for this algorithm has been implemented in Suade[12] [224] and works in conjunction with the ConcernMapper[13] [184] Eclipse plug-in, which programmers use to define the initial set I.

Saul et al. [197] developed an approach for recommending program elements that are relevant to a maintenance task (i.e., a feature) by approximating a random-walk of a system's call graph. The goal of their work is to find other methods that are related to a method by relying only on structural, call graph information. In their FRAN (Finding with RANdom walks) algorithm, a large set of related program elements are identified and ranked. Candidate related program elements are identified if they are on the same "layer" as the relevant program element, meaning they call or are called by the same methods. Relevance information about those program elements is then assigned using the HITS web mining algorithm [113]. An evaluation on Apache httpd shows that the FRAN algorithm improves upon the performance of Suade.

While ASDGs and topology analysis use both control and data dependencies to some extent, Trifu [214] introduced an approach to feature location based only on dataflow. The

[12] `http://www.cs.mcgill.ca/~swevo/suade/`
[13] `http://www.cs.mcgill.ca/~martin/cm/`

technique identifies the implementation of the functionality needed to produce a certain set of values called information sinks. A programmer defines the information sinks as a starting point and then all other variables that contribute to them are found by tracking dataflow dependencies, making this approach more fine-grained than most other feature location techniques. Tool support for the approach is provided by CoDEx, and a case study was performed on JHotDraw. Variables with no outgoing dataflow paths were automatically identified as information sinks, and the tool grouped 6,049 variables into 310 concerns (features). The approach was improved with the introduction of information sources [215], which define a boundary for a concern.

2.2.3 Textual Feature Location

Textual information embedded in source code comments and identifiers provides important clues about where features are implemented. This type of information has been used in many approaches for feature location in three mains ways: textual search with grep [157], information retrieval [45, 85, 142, 165], and natural language processing [103, 201]. The articles that introduce an approach to textual feature location are summarized below.

One simple and straightforward way in which programmers often search for source code that is relevant to their task is by using a textual search. They formulate a query that describes what they are looking for and then use a tool such as grep to find and investigate lines of code that match the query. Petrenko et al. [157] developed a feature location technique based on grep and ontology fragments. The ontology fragments record programmers' knowledge of a feature. As programmers gain more knowledge of the system, the ontology fragments can grow and be extended. Petrenko et al. hypothesized that the use of ontology fragments would increase the effectiveness of query formulation which would also increase the effectiveness of feature location. Case studies on Eclipse and Mozilla showed that ontology fragments required, on average, nine methods in Mozilla and ten methods in Eclipse to be inspected. These results were comparable to other feature location techniques [130, 159] in which programmers also only had to examine about ten methods.

Instead of pattern matching with grep, more sophisticated methods such as information retrieval can be used. Marcus et al. [134] employ Latent Semantic Indexing [59] to locate

features in source code. LSI is an advanced information retrieval technique that analyzes the relationships between words and passages in large bodies of text. LSI is applied to source code by extracting all identifiers and comments to form a corpus. As is common is most IR approaches, compound identifiers are split following observed naming conventions, and both the original identifier and its separate parts are added to the corpus. The corpus is partitioned into documents representing all terms found within a program element. Documents can be of different granularities such as classes or methods. The corpus is then transformed into an LSI subspace through Singular Value Decomposition (SVD). After SVD, each document in the corpus has a corresponding vector. To search for code relevant to a feature, a programmer formulates a query consisting of terms which describe the feature. The query is also transformed into a vector, and a similarity measure between the query vector and all the document vectors is used to rank documents by their relevance to the query. The similarity measure is known as the cosine similarity because it computes the cosine between the query and document vectors. The use of LSI for feature location was evaluated on the Mosaic web browser and compared to grep and ASDGs, and several advantages were found. LSI is almost as flexible as grep yet yields better results. Also, LSI was able to identify some relevant program elements missed by ASDGs.

Poshyvanyk and Marcus [165] introduce an extension to feature location with LSI that uses formal concept analysis (FCA) [83] to cluster IR-based results. FCA takes as input a matrix specifying objects and their associated attributes and then produces clusters, called concepts, of the objects based on their shared attributes. These concepts can be organized hierarchically in a lattice. In this case, the objects are methods and the attributes are words that appear in the source code of those methods. To combine the two types of analyses, LSI's results are automatically organized using FCA. The top k attributes of the first n methods ranked by LSI are used to construct FCA's input matrix and create a lattice. Nodes in the lattice have associated attributes (terms) and objects (methods), and programmers can focus on the nodes with attributes similar to their query to find feature-relevant methods. Through an evaluation of Eclipse, this approach was compared to LSI and found to be more efficient in terms of the number of methods programmers must consider before locating a relevant one.

Cleary and Exton [44, 45] also use information retrieval for feature location, but their contribution is to incorporate non-source code artifacts. Their approach, called cognitive assignment, considers indirect correspondences between query and document terms so that relevant source code can be retrieved even if it does not contain the query term. Queries are expanded by analyzing term relationships from both source code and non-source code artifacts. An extensive case study was conducted on Eclipse in which cognitive assignment was compared to language modeling [241], dependency language model [84], vector space model [196], and LSI, and cognitive assignment was found to be competitive with the other approaches.

The results of any textual feature location technique based on a query will only be as good as the query used. Often times, the query needs to be iteratively modified or refined. Gay et al. [85] introduce the notion of relevance feedback into textual feature location with IR. Relevance feedback incorporates user input to improve information retrieval results. After IR returns a ranked list of program elements relevant to a query, the developer rates the top n results as relevant or irrelevant. Then a new query is automatically formulated and new results are returned, and the process repeats. A case study was performed in which a single developer was asked to use IR and relevance feedback to locate the source code associated with change requests (representing features) in Eclipse, jEdit, and Adempiere[14]. Each change request had approved patches that had already been implemented in the system. The results indicate that relevance feedback is more effective and efficient than a pure IR-based approach.

Like information retrieval, Independent Component Analysis (ICA) [47] can examine source code text to identify features and their implementations [90]. ICA is a blind signal analysis technique that separates a set of input signals into statistically independent components. To apply ICA for feature location, a term-by-document matrix is constructed in which the rows correspond to methods, columns represent terms, and cells contain the frequency of a term in a method. ICA factors the matrix into two new matrices. The first new matrix, called the source signal matrix, stores independent signals which can be thought of as features. The second new matrix, the mixing matrix, holds information about

[14] `http://sourceforge.net/projects/adempiere/`

how relevant each signal is to a method. Unlike feature location with LSI, feature location with ICA does not need a query for a specific feature since it seeks to identify multiple independent signals (features) at once.

Textual analysis is not limited to information retrieval. Shepherd et al. [201] employ natural language processing. They observe that in source code, actions are represented by verbs, and nouns correspond to objects. Their technique, implemented in a tool called Find-Concept, leverages information about the use of verbs and their direct objects (nouns) in source code identifiers to create a natural language representation of the code called an action-oriented identifier graph (AOIG) [204]. In the AOIG, verb-direct object pairs are mapped to each of their occurrences in the code. The approach has three main steps: initial query formulation, query expansion, and a search of the AOIG. First, a user creates a query consisting of a verb and a direct object. Then, Find-Concept expands the query using NLP and its knowledge of the terms used within the software's source code to recommend new queries. Once the user refines the query, the tool locates nodes in the AOIG that contain a verb and direct object from the query and returns the methods to which they are mapped. Find-Concepts uses program analysis to identify any dependencies between the methods returned by the AOIG search and then presents the user with a visualization of the results as a graph. In a user study, Find-Concept's verb-direct object approach was compared to lexical searches using Eclipse and Google Eclipse Search [166] on a suite of open-source Java systems. Overall, Find-Concept was found to be the most effective search technique.

Hill et al. [103] also use NLP and the idea of query expansion and refinement in their approach to feature location based on contextual searching. Instead of focusing on verbs and direct objects, their analysis centers on three types of phrases: noun phrases, verb phrases, and prepositional phrases. They extract phrases from method and field names and generate additional phrases by also looking at a method's parameters. Once the phrases are extracted, they are grouped into a hierarchy based on partial phrase matching. The phrases are linked to the source code from which they were extracted. A user looking for a particular feature formulates a query and the tool searches the extracted phrases for matches. The result returned to the user is a hierarchy of phrases and the method signatures associated with them, giving some context to the results. This approach was

compared to Shepherd et al.'s [201] on the same software systems. Contextual search has been shown to significantly outperform the verb-direct object approach in terms of effort (number of queries needed) and effectiveness (f-measure).

2.2.4 Combined Dynamic and Static Feature Location

The combination of dynamic and static analysis is a well-known and powerful combination in other areas of research such as testing and program analysis [60, 79]. Feature location researchers have also made use of this combination in their own work. Dynamic analysis can be used to reduce the search space to only those program elements that were executed in a trace, and then static analysis can work on the smaller set of program elements to rank them or find additional relevant elements.

Eisenbarth et al. [73, 74, 75, 76] use FCA to cluster the information collected from dynamic information. FCA's input matrix is composed from execution traces. The objects are methods and the attributes are the features invoked during an execution scenario. After FCA is performed, the resulting concept lattice can be interpreted to identify candidate program elements that are solely relevant to a feature or contribute to a feature but are also used by other features. The program elements located by FCA are only a starting point, and programmers seeking additional relevant code can follow an approach similar ASDGs [39]. Eisenbarth et al.'s approach was evaluated at the method-level; Koschke and Quante [118] extended this work to locate features at the level of basic blocks.

While Koschke and Quante combined dynamic and static analyses in an approach that is more fine-grained than methods, Rohatgi et al. [186, 187] combine the two at a coarser level of granularity: classes. Their approach uses an execution trace and a class or component dependency graph (CDG) for feature location based on impact analysis. Distinct classes are extracted from a feature-specific execution trace, and then the CDG is used to rank the classes by the impact a change to them would have on the software system. The classes with the least amount of impact are most likely related to the feature. In an evaluation on Weka[15], a machine learning tool, the approach was able to identify and highly rank classes noted in the system's documentation.

[15] `http://www.cs.waikato.ac.nz/ml/weka/`

2.2.5 Combined Dynamic and Textual Feature Location

Dynamic and textual analyses are very synergistic when it comes to their use in feature location. Dynamic analysis generally yields good recall, while textual analysis has good precision. Their combination may lead to better results on both fronts. Both analyses can be used to rank program elements by their relevance to a feature, so a logical next step is to combine both of the rankings produced by these techniques. Another rational combination of dynamic and textual analyses is to use dynamic analysis to filter the program elements for textual analysis instead of ranking all the program elements in a software system.

PROMESIR (**P**robabilistic **R**anking of **M**ethods Based on **E**xecution **S**cenarios and **I**nformation **R**etrieval) [159, 160] performs feature location by combining "expert" opinions from two existing feature location techniques: SPR [5] and information retrieval with LSI [142]. The two approaches both rank program elements according to their relevance to the feature of interest. Those rankings are combined through an affine transformation to produce PROMESIR's results. The weight given to SPR or LSI can be varied to reflect the amount of confidence that should be assigned to each of the experts. Case studies performed on Eclipse and Mozilla show that PROMESIR typically outperforms the two techniques on which it is based.

Like PROMESIR, the SITIR (**Si**ngle **T**race + **I**nformation **R**etrieval) [130] approach to feature location is to combine execution information and IR, but only a single execution trace is collected for a feature. Then using a query relevant to the feature and LSI, only executed methods from the trace are ranked by their similarity to the query instead of all methods in the system. In case studies on jEdit and Eclipse, SITIR generally ranked relevant methods higher than LSI [142], SPR [5], or PROMESIR [160]. Liu et al. [129] developed a variant of SITIR called TAG, short for **T**r**A**ce + **G**rep. TAG performs tracing first and uses grep instead of LSI with the reasoning that grep is more lightweight. Liu et al. replicated SITIR's case studies with TAG; however, the results could not be compared directly because TAG's output is not ordered.

2.2.6 Combined Static and Textual Feature Location

Several researchers have combined static and textual analyses for feature location. This combination is a natural choice because either textual analysis can be used to reduce the overestimation that static analysis is prone to or static analysis can be used to find additional candidate program elements given a starting set of highly relevant ones from textual analysis. Thus, uniting these two types of analysis has the potential to yield better results than either static or textual analysis alone.

SNIAFL [243, 244] is a **s**tatic, **n**on-**i**nteractive **a**pproach to **f**eature **l**ocation. SNIAFL uses information retrieval in conjunction with a branch-reserving call graph (BRCG), essentially an expanded version of a call graph with branch information. An initial set of program elements (methods) specific to the feature is located using information retrieval, and then additional relevant elements are found using the BRCG. The initial set is produced by using the vector space model [196] to obtain and rank methods by their similarity to a query. A gap threshold technique is used to find the largest difference between the similarities of consecutively ranked methods. The methods above this gap are considered to be the initial elements specific to the feature. From the initial set, the BRCG is pruned to remove branches that are not in the initial set. Also, the relevance of branches that are included in the initial set is propagated through the graph's dependencies, essentially generating a static pseudo-execution trace. In case studies on two GNU software systems, SNIAFL had better precision and recall than both a pure IR approach and a purely dynamic approach, lending evidence to the fact that combining static and textual analyses is more successful than using them as standalone techniques.

Like SNIAFL, Dora [102] combines static and textual analysis to perform feature location. Programmers formulate a query which is used to compute a method relevance score that is based on the term frequency-inverse document frequency of words that appear in methods' names. Then, starting from a set of seed methods defined by the programmer, Dora follows static caller/callee edges to identify additional relevant methods using the relevance score. Dora was evaluated on a number of open source Java systems and compared to Suade and two naïve textual and static approaches. The benchmark for these systems

was determined by a user study [183] in which programmers were asked to locate the implementations of several features. Dora was found to be the most successful technique in the evaluation.

In Dora and SNIAFL, one type of analysis is used to prune another. Shao and Smith [200] combine information retrieval and static control flow information in a different manner for feature location. First, LSI is used to rank all the methods in a software system by their relevance to a query. Then, for each method in the ranked list, a call graph is constructed. A method's call graph is inspected to assign it a call graph score. The call graph score counts the number of a method's direct neighbors that also appear in LSI's ranked list. Finally, the method's cosine similarity from LSI and its call graph score are combined using an affine transformation, and a new ranked list is produced. This technique has only been evaluated in one case study where it was compared against LSI on a C++ program called iVistaDesktop, which simulates Microsoft's Windows Vista operating system. The study showed that this approach ranked the one relevant method of a change request first, while LSI ranked it seventh.

Ratiu and Deissenboeck [169, 170] use ontologies to recover the mapping between source code and real-world concepts. Their approach is not explicitly aimed at feature location but at linking program elements to concepts, which could be features. They developed a framework that describes semantic defects caused by improper naming and an algorithm to recover the mappings between ontology elements and program elements. The algorithm maps concepts and program elements via graph matching. Concepts are graphed in an ontology and programs are represented by a UML-like dependency graph. The framework and algorithm have been applied to the Java standard library, finding actual examples of semantic defects.

2.2.7 Combined Dynamic, Static, and Textual Feature Location

Cerberus [62] is a feature location technique that utilizes three types of analysis: dynamic, static, and textual. Currently, it is the only approach that leverages all three types of analysis. At the core of Cerberus is a technique called prune dependency analysis (PDA), whereby a relationship between a program element and a feature exists if the program

element should be removed or modified if the feature were to be pruned from the software system. Given an initial set of relevant elements to be pruned, PDA infers additional relevant elements. Cerberus uses PROMESIR to combine rankings of program elements from execution traces with rankings from information retrieval to produce seeds for PDA. Cerberus' authors created a large benchmark for Rhino[16], an open source Java implementation of Javascript, in which the code for over 400 features defined in the system's documentation were manually located. This benchmark was used to evaluate and compare Cerberus to software reconnaissance [229], SPR [5], DFT [77], LSI [142], finding that combining the three types of analysis was the most effective approach.

2.2.8 Other Feature Location Techniques

Only four feature location techniques surveyed do not rely on dynamic, static, or textual analysis. Instead, they use other types of analysis and sources of information to locate features. One looks at a developer's program exploration behavior, while the others utilize historical, archived information. The use of alternative types of analysis in conjunction with dynamic, static, and textual analyses remains an open issue.

Robillard and Murphy [180] propose a unique approach to feature location that automatically analyses a transcript of a program investigation session in an integrated development environment. The transcript records which program elements were visible to a developer during a maintenance task and how they were accessed: through a code browser, following a cross-reference, recalling an open window or tab, scrolling, or keyword search. For each event in the transcript, all visible program elements are determined. Then, for each visible element, a probability that it is the element in which the programmer was interested is assigned to it. The probabilities are based on weights associated with each event type. Next, a correlation metric is calculated between all pairs of program elements. The correlation is based on how closely two elements were accessed in the transcript. Finally, concerns (features) are generated by clustering program elements, and the concerns can be named and saved for later retrieval.

[16] `http://www.mozilla.org/rhino/`

CVSSearch [38] is an approach and tool for feature location that searches for source code by using CVS log comments. CVS comments generally describe the change made to the lines of code which are being committed, and those comments typically hold true for many future revisions of the software. The tool maps CVS comments to their corresponding revision and then examines the changes between consecutive versions to map source code to comments. A user can enter a query, and CVSSearch[17] returns all lines of code whose comments contain at least one of the query words. Each returned line also has a score indicating how well it matches the query.

Like CVSSearch, Robillard and Dagenais [178] also use historical information from a repository for feature location. They use change history to identify clusters of program elements related to a task (i.e., a feature). Given a query of a set of program elements, their approach groups repository transactions by the number of nearest-neighbor program elements they share and returns a cluster of elements related to the query. Various filtering heuristics can be applied to the results to remove program elements that are unlikely to be related. For instance, if a program element is modified in a high percentage of all of the transactions in the repository, it can be ignored. The approach was evaluated on 12 years of change data for seven open source system and found that only a small fraction of changes would have been helped by change clusters.

Hipikat [217, 218] is a feature location approach that also makes use of archival information for feature location, but instead of identifying candidate program elements, Hipikat recommends artifacts from a project's archives such as online documentation, versions, bugs, or communications. Hipikat forms a group memory [219] from a project's history as recorded in source code repositories, issue trackers, communication channels, and web documents. Links between these artifacts are inferred using IR. For example, a source code version can be linked to a bug report if the bug's id is included in a repository commit log message. This history is used to find relevant artifacts in response to a user query. The query consists of an artifact, potentially a program element, for which the user wants recommendations of related artifacts. Hipikat responds with a list of artifacts ranked by their relevance. The tool has been used in two case studies. In the first, Hipikat was validated on

[17] `http://cvssearch.sourceforge.net/`

AVID[18], and in a second, it was used to aid programmers performing a change to Eclipse.

2.3 Feature Location Tools and Studies

In addition to the many research articles that introduce feature location techniques, there are numerous articles describing feature location tools, case studies, industrial studies, and user studies. This section summarizes these tools and studies.

2.3.1 Tools

Tool support for feature location removes much of the manual burden associated with searching for a feature's program elements. In addition to providing an overview of existing feature location techniques, this survey also describes tools that can be used for feature location. Some of the techniques summarized in Section 2.2 have prototype tools that are not available; therefore they are not listed here. Also, some tools are not directly associated with any particular approach, but they can be used for feature location, to document features, or program exploration, so they are included here.

2.3.1.1 Tools for Dynamic Feature Location

Some of the earliest feature location techniques relied on dynamic analysis since features are typically visible during the execution of a software system. Feature location tools take advantage of this visibility.

RECON, RECON2, and RECON3[19] are tools that implement the software reconnaissance [229] approach to feature location described in Section 2.2.1. Wilde and Casey [227] report on applying RECON to industrial software. In their study on using software reconnaissance for program exploration, Wilde and Casey found the tool to be very selective because it never marked more than 13 methods for a feature. They also observed that the tool seemed to find code that was near relevant program elements. In a second part of their study, they examined using software reconnaissance for traceability to build a large mapping of multiple features to code. The tool was used to run a large set of test cases that were

[18] `http://people.cs.ubc.ca/~murphy/AVID/`
[19] `http://www.cs.uwf.edu/~recon/`

marked with the features they exhibited, and then the collected traces were analyzed to find traceability relations that mapped features to code. With this traceability knowledge, a programmer modifying a program element is aware of the other features implemented in that program element.

Ibrahim et al. [105] also report on their experiences applying RECON2 the Generate Index (GI) project. Their findings echo the conclusions of the previous study. Software reconnaissance is based on test cases, but selecting appropriate scenarios to execute can be difficult. However, only a few test cases are generally needed for a feature. After the analysis, software reconnaissance is good a locating a starting point for feature location, but further investigation for additional relevant program elements should be performed.

STRADA (**S**cenario-based **TRA**ce **D**etection and **A**nalysis) [69] is a tool to help developers uncover the mappings between features and code during testing. It is based on Egyed's trace analysis research [67, 70, 71, 72]. Given a set of test cases for a feature, STRADA observes the code that is executed during testing, initially identifying all the executed code as relevant to the feature. However, since not all of the invoked code actually pertains to the feature, STRADA analyzes the traces using logical constraints to exclude irrelevant program elements. The tool visualizes its knowledge of feature-to-code mappings in a matrix. It has been evaluated on ArgoUML[20], GanttProject[21], and a video-on-demand player[22].

2.3.1.2 Tools for Static Feature Location

Static feature location tools analyze the dependencies and relationships among program elements, similar to the way a developer might explore a program. The two tools discussed here realize the ASDG and topology analysis static feature location techniques.

Ripples [40] is a tool that implements the ASDG approach to feature location. The tool extracts an ASDG from C code and visualizes it for the programmer who can mark relevant nodes. JRipples[23] [35] is a similar tool that supports the approach for Java source code in

[20] `http://argouml.tigris.org/`
[21] `http://www.ganttproject.biz/`
[22] `http://peace.snu.ac.kr/dhkim/java/MPEG/`
[23] `http://jripples.sourceforge.net/`

Eclipse, except without the visualization. Both tools can also be used for impact analysis and change propagation by tracking and monitoring the status of program elements.

Suade [224], an Eclipse plug-in, is another tool that statically performs feature location. It implements the topology analysis [176] approach discussed in Section 2.2.2. Suade has been used in a case study comparing several program exploration tools [56], and it has also been directly compared to Dora [102], another static feature location technique.

2.3.1.3 Tools for Textual Feature Location

Textual feature location tools search for relevant program elements based on a user query. The standard utilities for searching source code are grep or an integrated development environment's built-in search functionality. The tools presented here go beyond these basic search techniques by employing information retrieval.

Google Eclipse Search[24] (GES) [166] is an Eclipse plug-in that facilitates efficient source code searching and browsing by integrating Google Desktop Search (GDS)[25] and Eclipse. GDS is an off-the-shelf component that uses information retrieval. It allows users to search for files on their computer similar to they way they would search for information on the Internet by issuing a query. By integrating GDS with Eclipse, programmers can search source code in a similar fashion. One advantage of using GDS is it unobtrusively re-indexes the search space when the source code changes. Also, compared to Eclipse's file search functionality, GES is considerably faster.

IRiSS [163] and JIRiSS [162] are both tools for textual feature location. IRiSS implements information retrieval-based feature location as an add-on for MS Visual Studio .NET, while JIRiSS is an Eclipse plug-in. Both tools work like a development environment's built-in search functionality, but instead of only displaying the lines of code that match a query, those lines' corresponding classes and methods are also listed. This allows a programmer to sort the results by different levels of granularity and to visit the classes or methods with the most matches. Also, since IR is used, the results returned for a query can be ranked by their relevance. JIRiSS is an extension to IRiSS that also includes fragment-

[24] `http://ges.sourceforge.net/`
[25] `http://desktop.google.com/`

based searches, software vocabulary extraction, query spell checking, and word suggestions to improve queries.

2.3.1.4 Tools for Documenting Features

Once a feature's relevant program elements have be found using feature location, they should be saved so that the search does not have to be repeated in the future. Features and their relevant program elements can be documented in ConcernGraphs [179, 182], a model that describes which program elements pertain to a feature. Tool support for ConcernGraphs is provided by FEAT[26] [181], the **F**eature **E**xploration and **A**nalysis **T**ool, as well as ConcernMapper [184]. ConcernTagger[27] extends ConcernMapper with the ability to compute a number of concern-specific metrics. The **F**eature **L**ocation and **T**extual **T**racing Tool[28] (FLAT^3) [198] also extends ConcernMapper by adding automated support for textual and dynamic feature. FLAT^3 is discussed in detail in Chapter 6. In each of these tools, programmers can define and name features and then associate entire or partial classes, methods, and fields with them. The tools, except for FLAT^3, leave feature location as a manual task and focus on documenting features and their related elements once they are found. However, once the features and their program elements are documented, they can be saved and retrieved at a later time, thus avoiding the need to repeat searches.

2.3.1.5 Visualization Tools

Visualization tools for feature location either allow for the exploration of dynamic information or highlight candidate program elements in source code. The visualizations generally create an abstracted global view of the system in which relevant program elements are emphasized.

TraceGraph [133] is a feature location tool that allows for the visualization of execution traces. As a software system is running, TraceGraph analyzes the execution and visualizes which program elements were invoked during a time interval. The visualization is essentially a matrix in which the rows represent program elements, the columns correspond to

[26] `http://www.cs.mcgill.ca/~swevo/feat/`
[27] `http://www.cs.columbia.edu/~eaddy/concerntagger/`
[28] `http://www.cs.wm.edu/semeru/flat3/`

time intervals, and the cells indicate if the program elements were called during that time interval or not. Additionally, the first invocation of a program element is highlighted in the visualization. TraceGraph was evaluated on the Mosaic web browser as well as the Joint Surveillance Target Attack Radar Subsystem (Joint STARS), a proprietary system developed by Northop Grumman for the United States Air Force. The tool's visualization was useful for feature location because it emphasized the first time an element was called, which often corresponded to a feature being triggered. TraceGraph was also applied in an industrial case study on feature location [207] where it was used for trace differencing and identifying code uniquely executed by a feature, and in another study on distributed simulation software [230].

Like TraceGraph, the prototype tools created by Bohnet and Döellner [19, 20, 21, 22, 23, 24] also visually explore dynamically extracted information, but in this case, as a call graph. Since a call graph can be large, in order to reduce the search space for the user, the tools provide cues to identify code relevant to the feature of interest. The tools also provide a number of different types of visualizations. In one prototype, a graph exploration view shows other methods that pass control flow to or receive control flow from a given method. In this view, the tool only shows methods in a neighborhood if they are judged to be relevant based on execution time, while another tool has textual and 3D landscape views. These tools effectively extract dynamic call graph information and guide programmers during navigation.

Instead of relying on dynamic information, AspectBrowser [96] is a tool that assumes that features follow the idea of information transparency [95]: design decisions that cannot be encapsulated in a single module use a common signature or terminology that can easily be exploited by search tools. The AspectBrowser tool[29] allows users to search a code base using pattern matching and then visualizes the results in two ways. All query matches can be highlighted in the source code, and the programmer can browse to find them. Alternatively, programmers can use a global view to see how a feature is scattered throughout the system. In the view, each line of code is represented by a row of pixels, and highlighted rows indicate lines of code that match the query. Multiple search results can be viewed at once

[29] `http://cseweb.ucsd.edu/~wgg/Software/AB/`

to understand the interaction between several features.

Xie et al. [235] also developed a suite of tools that visualize based on textual analysis, but instead of grep, they use information retrieval. IRiSS [163] performs feature location via IR. Then, sv3D (source viewer 3D) [135] creates 3D renderings of the results, showing poly-cylinders that represent program elements. The colors of the poly-cylinders correspond to the elements' similarity to the query following a pre-defined color scheme. The height of the poly-cylinders represent browsing history, so the taller the cylinder, the more times the program element was visited in the past.

2.3.1.6 Program Exploration Tools

Program exploration tools support developers when performing a variety of maintenance tasks. Since feature location is central to many maintenance activities, program exploration tools can be used for feature location as well.

JQuery [107], an Eclipse plug-in, is a source code browsing tool designed to help programmers when dealing with features that have scattered implementations. The tool combines navigation based on relationships (as in a hierarchical browser) with the flexibility of query languages. Program exploration in JQuery begins with a query and a list of variables. The query determines which elements to show in the browser, and the variables establish how to organize them into a tree. The query defines the type of program element to search for given some parameters such as its name or a type of relationship. The results of the query are returned in a hierarchical tree, and users can further explore the tree with additional queries that expand nodes into sub-trees. The tool aims to reduce the burden of program investigation on developers. It helps them remain oriented by not having to switch between multiple views, and it records their exploration path in the tree format.

Ferret [55] is a tool for answering conceptual queries, which are questions about a software system a programmer may have. The model Ferret is based on supports the composition and integration of different sources of information into a queryable knowledge-base. A source of information is known as a sphere, and examples include structural relationships in source code, dynamic call information from an execution trace, and revision history recorded in a software repository. Ferret supports 36 different conceptual queries

such as "What calls this method?", "What are this class' subclasses?", "What are all the fields declared by this type?", and "What transactions changed this element?" These types of queries represent questions programmers may have when investigating a software system in order to locate a feature's implementation.

De Alwis et al. [56] performed a comparative study of three program exploration tools: JQuery [107], Ferret [55], and Suade [224]. Eclipse was used as a baseline for comparison. They hypothesized that programmers would find it easier to work with a tool, need to examine less code as compared to using Eclipse, and generally gain a better understanding of the task at hand. The participants in the study were 18 professional programmers, and they were asked to investigate two change tasks in jEdit. In the first task, they used only Eclipse, and in the second task, they used one of the exploration tools. The order of the tasks and choice of tools was randomized. An instrumented version of Eclipse captured all events the programmers performed during their investigation. Additionally, the participants recorded the relevant elements they found in an Eclipse view built for the study. The NASA Task Load Index (TLX) [99] was used to assess task difficulty, and distance profiles were used to gauge the degree to which the participants remained on-task. The TLX scores showed no difference in task difficulty that could be attributed to using a tool or not. Similarly, the distance profiles did not indicate that the tools had any strong effect on the tasks. Overall, the authors concluded that program exploration tools had little effect on the tasks and that individual programmers' strategies caused them to be more or less efficient.

2.3.2 Case Studies

A number of case studies involving feature location have been performed, ranging from comparisons of existing techniques, industrial case studies, and user studies. Each type of study is useful. Comparisons evaluate several feature location techniques on the same systems and features, making it easier for researchers and practitioners to understand the strengths and weaknesses of each approach. Industrial case studies show the applicability of an approach in non-trivial settings. Finally, user studies provide insight into how programmers understand and search for code, and these insights can be incorporated into

feature location techniques and tools.

2.3.2.1 Case Studies Comparing Feature Location Techniques

Early feature location techniques were applied when procedural programming was the predominant paradigm. After object-oriented programming gained popularity, Marcus et al. [141] studied whether feature location was still needed since object-oriented code is supposed to be structured such that classes implement singular concepts. They compared the performance of three static feature location techniques: pattern matching with grep, a depth-first dependency search [39], and information retrieval using LSI [142]. Three programmers, each assigned to a different technique, participated in a case study to locate features in Art of Illusion[30], a 3D modeling studio written in Java, and in Doxygen[31], a source code documentation generator written in C++. They concluded that object-orientation does not always allow for quick and easy identification of the program elements relevant to a feature. Therefore, feature location techniques are still needed for object-oriented systems.

When a new feature location technique is introduced, it is often directly compared with similar existing approaches as part of its evaluation. Some articles related to feature location focus solely on case studies comparing several techniques. Wilde et al. [225, 226] compare software reconnaissance [229], ASDGs [39], and grep in a case study to locate two features in legacy FORTRAN code. The system, CONVERT3, is part of a suite of geometric modeling programs and is used to convert models to formats required by other tools. For the study, three teams each used one of the feature location techniques to find the code for two features of CONVERT3. The software reconnaissance and ASGD teams were able to gain sufficient understanding of the source code, but the team using grep was not. The authors concluded that grep was the least reliable approach but it is very quick and can locate features that cannot be explicitly invoked dynamically. After grep, software reconnaissance was deemed to be the next fastest method of feature location. However, its results may not present a user with enough context to be comprehensible. The ASDG

[30] `http://aoi.sourceforge.net/`
[31] `http://www.stack.nl/~dimitri/doxygen/`

approach was the most difficult to apply but the most systematic and allows for the best understanding of the relevant code.

Revelle and Poshyvanyk [173] performed an exploratory study evaluating several feature location techniques that return ranked lists of program elements (methods). The approaches they compared were information retrieval (LSI-based feature location [142]), information retrieval plus dynamic analysis (SITIR [130]) , and information plus dynamic and static analysis (similar to Cerberus [62]). For IR, they assessed user-formulated queries as well as method seed queries in which the text of a method already known to be relevant to a feature was used as the query. For dynamic analysis, they used both full execution traces and marked traces in which only the portion of a system's execution when a feature is invoked was traced. Dynamic analysis was combined with IR by pruning unexecuted methods from the ranked list. When all three types of analyses were combined, a program dependence graph was traversed starting from a seed by following dependencies only if they were executed and had textual similarities to the query that were above a given threshold. Most feature location techniques that return a ranked list are evaluated in terms of where the first relevant element appears on the list. This case study aimed to evaluate these approaches in terms of how well they find near-complete implementations of features, meaning how well they find as many relevant program elements as possible. Their conclusions were that none of these approaches perform particularly well in that regard since feature location is usually used to find a starting point and impact analysis tools are used to find more complete implementations. They did note that marked traces generally outperformed full traces and that the method seed queries, which can be automatically generated, performed just as well at user formulated queries. More details on this exploratory study are given in Chapter 3.

2.3.2.2 Industrial Case Studies

Most feature location case studies focus on open source software. However, case studies carried out on industrial software give a sense of a technique's real world applicability. Unfortunately, only a few such studies have been performed, and more are needed. As previously discussed, TraceGraph [133] was used in an industrial setting [133], and Wilde

et al. [226] compared a number of approaches on industrial software. In addition to these studies, Van Geet and Demeyer [221] report on their experiences of applying Eisenbarth et al.'s [76] formal concept analysis of execution traces feature location technique in an industrial setting. The context was a pre-study phase for the migration of a banking system written in COBOL. Scenarios for two features were executed using a web interface, and three separate iterations of the approach were conducted. Each iteration attempted to reduce the number of modules considered by using different combinations of scenarios that did and did not invoke the feature. A domain expert provided the modules relevant to each feature for evaluation purposes, and in two out of three iterations of the approach, all the relevant modules were in the generated concept lattice. Three additional relevant modules were also identified that had not previously been named by the domain expert.

2.3.2.3 User Studies

Studies that focus on how programmers search and comprehend source code are important to feature location research. These types of studies provide insights into how developers find a feature's implementation or gain understanding of a system. In turn, these insights can be incorporated into feature location research in order to develop approaches that are organic and easy for programmers to use. Four relevant user studies are discussed below, and while this is not an exhaustive list of user studies related to feature location, even more studies are necessary to advance the state of the art.

LaToza et al. [123] performed a user study in which 13 participants worked for three hours on understanding and improving the design of two features in jEdit. The participants' activities were recorded using think-aloud, video, and Eclipse instrumentation. The goal of the study was to answer questions about how programmers' experience affects the changes they make to code, how it affects how they work, and how they reason about design during coding tasks. LaToza et al. found that the more experienced programmers addressed the causes of problems, beginners focused on the symptoms, and that the experienced programmers identified relevant methods and implemented changes more quickly than the novices. They also discovered that the participants' activities centered on fact finding. The programmers sought facts relevant to their task, so they investigated certain methods and

learned facts about the software system, and as they learned enough facts, they were able to propose design changes. Therefore, feature location techniques should not only help identify relevant program elements, but they should also aid in fact finding and program comprehension.

Robillard et al. [177] also conducted a study on how programmers explore source code when performing a change task. Five programmers were asked to modify jEdit so that users can explicitly disable the autosave functionality. They were also given five requirements for their solution. The data collected included artifacts produced or modified by the participants as well as video recordings of their screens. The programmers' success was judged in terms of time to complete the task and by how many of the task's requirements they successfully implemented. Robillard et al. analyzed the behavior of each participant by transcribing the screen videos into events. Each event recorded the time it occurred, the method being examined at that time, how the method was accessed (scrolling, browsing, searching, etc.), and whether the method was modified. Based on their observations, they concluded that a methodical, ordered investigation of a system's source code is more effective than a systematic, opportunistic one. They found that programmers should follow a plan when exploring a program, that they should perform focused searches in the context of their plan, and that they should keep a record their findings. Based on these findings, feature location techniques should facilitate orderly program exploration.

Revelle et al. [172] undertook two exploratory studies on how programmers identify features and their implementations in source code. In the first study, the features of GNU sort[32] plus their relevant source code were found manually by one programmer and then compared to those of Carver and Griswold [36]. In the second study, two programmers manually located features and their implementations for a Java implementation of the Minesweeper game. Revelle et al. compared the actual concepts recognized as features as well as the code associated with those features, looking for common trends in how developers identify and locate features. Based on their observations, they developed a set of guidelines for how to recognize the existence of a feature and how to record a feature's associated code in a tool called Spotlight [50]. The guidelines suggest relying on both static and textual

[32] `http://www.gnu.org/software/coreutils/`

information and mapping features to program elements at various levels of granularity.

Ko et al. [114, 117] performed an exploratory study to investigate developers' strategies for understanding unfamiliar code. Ten participants worked using Eclipse on five maintenance tasks associated with the Paint[33] application. Screen-capture videos recorded the developers' work during the study. To simulate a more realistic working environment, the programmers were interrupted every 2.5 to 3.5 minutes and required to answer a multiplication question. Monetary incentives were offered for correctly completing the tasks, and penalties were inflicted for ignoring or incorrectly answering the multiplication questions. The study found that programmers interleave three activities when exploring source code: searching for relevant code either manually or with tools, following the dependencies of found relevant code, and collecting relevant code and information in Eclipse's interface (i.e., package explorer, tabs, and scroll bars). However, searches often failed, Eclipse's navigation tools imposed overhead when following dependencies, and developers lost track of relevant code in the interface. On average, 35% of a developer's time was spent reviewing search results and on navigation. Based on the observations of this study, the Ko et al. make a number of suggestions for future tool development. First, tools need to provide better relevance cues so programmers do not miss important code or misinterpret irrelevant code. Second, dependency searches need to be more practical, such as by highlighting the dependencies of the currently selected program element. Third and finally, programmers need a better way to collect, organize, and view the relevant information they find, such as being able to see all relevant information for a given task at once. These recommendations may help programmers find task-relevant code more quickly and efficiently and were used in the design of a new debugging tool [116].

2.4 Discussion and Open Issues

Feature location is an essential aspect of many software maintenance tasks, and because it can be challenging to perform manually, researchers have introduced many techniques to lessen the burden of searching for a feature's relevant code. Even with these numerous

[33] `http://www.cs.cmu.edu/~marmalade/studies.html`

approaches and advancements, open issues remain in the field of feature location. One question that remains unanswered is, "What is the best way to perform feature location?" This question cannot be easily answered without an extensive comparison of analysis-specific issues and a comparison of all existing approaches. Such a comparison could be facilitated by well-established benchmarks. Currently, there is no commonly accepted set of features mapped to the code that implements them that could be used to compare feature location techniques. Such a benchmark is needed in the research area. Additionally, while there are various techniques that support feature location, not all approaches have publically available tools, and the tools that are available do not support both locating and documenting a feature's implementation. Other open issues are usability studies of feature location techniques and integrating feature location into software engineering courses. The remainder of this section discusses these open issues and their associated avenues for future research. While this discussion brings to light these important topics, more panels and workshops, like the one on the identification of concepts, features, and concerns in source code held at the International Conference on Software Maintenance in 2005 [140], are necessary to resolve many of these issues.

2.4.1 Comparisons

Given the wide variety of existing techniques, developers that need to perform feature location have many options, but which approach is the best for a specific situation? What parameters should be used for a certain type of analysis? Which type of analysis yields the best results, or is a combination of analyses the best? Some case studies have been performed comparing multiple feature location techniques [173, 225, 226], but they only have a few data points, which impedes the ability to draw statistically significant generalizations from their results. These studies are also limited in the number of examined approaches, focusing on a subset of approaches that present results in a similar fashion. An obstacle to comparing techniques is the presentation of their results. How does one evaluate one result set that ranks program elements to another that does not? Determining the best way to directly compare the performance of feature location techniques remains an open issue.

Not only does there need to be a comparison of techniques based on different types of

analysis, but there also needs to be an evaluation of the best configuration of each type of analysis. For instance, dynamic analysis has many possible options for collecting traces. The granularity of execution traces can be classes, methods, or even lines of code. In addition, the entire execution can be logged from start up to shut down, or only select portions of the run can be captured. With static analysis, like with dynamic, granularity is also a parameter. Additionally, the type of dependencies (control or data) to take into account is another issue to consider. With textual analysis, preprocessing options such as stemming and stop word removal are commonly used, but their effect on feature location has not been fully studied. Also, textual analysis can be achieved through information retrieval methods or through natural language processing. While the varied IR methods have been compared, the effectiveness IR and NLP has not been compared in the context of feature location. A comparison would determine if the extra expense associated with NLP is worth the precision, or if the less expensive IR methods are sufficient. Thorough investigations comparing these different configurations of each type of analysis would reveal the most favorable settings for feature location.

There are many other open issues in feature location that could potentially be resolved through comparisons. The main types of analyses are dynamic, static, and textual, but historical analysis has also been used, but not in conjunction with any other analysis. It remains to be seen if combining historical analysis with any of the others is a viable approach to feature location. Just as three types of analysis comprise the majority of existing techniques, two programming languages dominate the area of feature location. Most existing approaches have been applied to Java or C/C++. However, feature location should branch out to support more languages so it can be applied to more software systems.

2.4.2 Benchmarks

The comparison of feature location techniques would be facilitated by the existence of benchmarks that could be used to consistently evaluate the approaches. Currently, there are a number of systems that have been used in the evaluation of many feature location techniques such as Eclipse, JHotDraw, jEdit, Mozilla, and Firefox, but the features used for the evaluation are not consistent. Even if two approaches are evaluated on the same

system, if different features are used, comparing the two techniques is difficult. Another problem with assessing feature location is in knowing the "gold set" of program elements that implement a feature. The most commonly used method for determining the source code that is relevant to a feature is to mine bug tracking systems. However, the code associated with a feature may be incomplete if a bug fix only touches part of a feature's implementation. In the presence of these issues, the field of feature location research needs to establish standards for validation. The best solution may be benchmarks that can be used to easily compare approaches. The benchmarks would consist of a set of features from a software system or several systems. Each feature would be mapped to the source code that implements it. Ideally, the benchmarks would have variable granularity so that it could be applied to approaches that identify relevant classes, methods, and variables. Robillard et al. [183] and Eaddy et al. [61] have taken a step in this direction, making available their data sets in which programmers who were not necessarily systems experts mapped features to methods and fields in open source Java applications. Still, well-established and complete benchmarks of systems from a variety of domains and languages will make the evaluation and comparison of feature location techniques easier.

2.4.3 Tools

Even though this survey encompasses many tools that support feature location, the majority of feature location techniques do not have a publically available tool, meaning programmers wanting to apply such an approach may need to recreate the technique's methodology. Additionally, some tools are useful for investigating a program and locating features (See Section 2.3.1), while other can be used to store the mappings between features and source code [181, 184], but currently only one tool does both [198]. Combining the functionalities of finding features' implementations and being able to save them is a logical next step for tool development. Finally, de Alwis et al.'s [56] found that existing tools have little effect on programmer's efficiency, so further research is needed to improve the effectiveness of tools.

2.4.4 User Studies

While there have been several user studies investigating how programmers search and explore source code during maintenance, these studies are based on a small number of users. Further studies are needed with more users to be able to derive conclusive results. Additionally, there has been a lack of studies examining usability aspects of feature location. Do existing feature location techniques reduce the amount of time and effort developers spend on maintenance? What are the practical benefits and costs of using different types of approaches? For instance, collecting execution traces for an approach that uses dynamic analysis requires overhead in terms of the time spent to develop scenarios or test cases and capture traces. Information retrieval involves indexing the source code of a software system, which can be time-consuming. Studies are needed to determine whether or not the overhead of collecting traces or indexing a corpus yields improved feature location results and is worth the cost.

2.4.5 Feature Location and Education

Given that feature location is such an extensive area of research and also an important part of software maintenance, it should be taught in software engineering courses at universities and colleges. Petrenko et al. [34, 155] have argued for the inclusion of software maintenance and evolution in software engineering courses along with traditional development. Teaching maintenance exposes students to more realistic experiences since in industry, 70% or more of programmers' time is devoted to maintenance [199, 209]. Feature location is a significant part of the maintenance phase since before changes can be made to a system, the relevant program elements must be found. Therefore, feature location should be introduced as a topic in software engineering courses to better prepare students.

2.5 Conclusion

Through a comprehensive examination of 88 feature location articles encompassing research, tools, and case, industrial, and user studies, this survey has presented a taxonomy that classifies the literature along nine key dimensions. The taxonomy facilitates the

comparison of existing feature location techniques and illuminates possible areas of future research. Researchers can use the taxonomy and survey as a basis for advancing the field, while practitioners can use it to identify techniques and tools that are well-suited to their needs. This survey has also shed light on open issues in feature location such as the need for comparisons and benchmarks. By structuring and organizing the research area of feature location, this taxonomy and survey contributes clarity to the field and should aid in resolving some of the open issues.

This chapter has given a comprehensive overview of existing work in feature location. The next two chapters cover our novel contributions to the area. Our work compares and expands on some of these existing techniques. Chapter 3 presents an exploratory study comparing feature location techniques based on combinations of dynamic, static, and textual analyses. Chapter 4 introduces new feature location techniques that incorporate web mining algorithms with textual and dynamic analyses.

Chapter 3

An Exploratory Study on Assessing Feature Location Techniques

Software maintenance and evolution tasks first require programmers to understand the implementation of specific parts of an existing software system [125]. To do so requires locating the source code that implements functionality, an activity known as *concept assignment* [12] or *feature location*. The previous chapter gave an overview of this research area and described existing feature location approaches. Most feature location techniques have been shown to be effective at finding a starting point of a feature's implementation, i.e., one method that is relevant to that feature [130, 160, 165]. However, it is rarely the case that a single method is the sole contributor to a feature. These techniques leave it up to programmers to find the other methods that implement a feature.

For feature location approaches to be truly effective, they need to find *near-complete* implementations of features. We define the term near-complete to denote a partial but close to total set of methods that implement a feature since knowing all the methods that implement a feature is rather subjective. One programmer may consider a method relevant, while another may not [183].

This chapter presents an exploratory study of ten feature location techniques that use various combinations of textual, dynamic, and static analyses. The approaches are evalu-

ated in terms of how well they locate near-complete implementations of several features in the jEdit and Eclipse software systems. As part of the assessment, we designed easy-to-follow guidelines for evaluating feature location techniques. Additionally, we explored a new mechanism for formulating queries used by textual analysis that automatically constructs a query from the identifiers of a method.

Our results highlight the challenge of feature location since no single technique was universally successful. We provide observations of situations when the approaches do and do not work well. One promising result is that our new means of automatically creating a query for textual analysis performs comparably to a query formed by a human. We used the results of this exploratory study to guide the development of new feature location techniques presented in Chapter 4.

3.1 Feature Location Techniques

A *feature* is a functional requirement of a program that produces an observable behavior which users can trigger. Examples include spell checking in a word processor or drawing a shape in a paint program. The term feature is intentionally defined weakly in the literature so it is suitable in many situations [5, 76].

Feature location is the activity of identifying the source code elements that implement a feature [12]. We investigate several approaches to locate the source code associated with a feature using textual, dynamic, and static analyses. Next, we explain each type of analysis and how we combined them in this work.

3.1.1 Core Techniques

Textual analysis. The implementation of a feature, even if dispersed among many methods, may use a consistent vocabulary in terms of identifiers and the words appearing in comments [95]. One approach to locate features is to determine textual similarities among a user query and source code elements (e.g., methods). A query is a set of words formulated by a user that describe a feature. Alternatively, a query can be automatically comprised of the identifiers and comments in a method that is known to be relevant to a feature. In

either case, textual analysis and feature location can be performed using the information retrieval technique known as Latent Semantic Indexing (LSI) [59]. With LSI, the relation between terms (words) and documents (methods) can be discovered. In brief, comments and identifiers are extracted to form a corpus. LSI indexes the corpus and creates a signature for each document (method), and these indices are used to define similarity measures between methods. Users can formulate queries in natural language (*nl-queries*) or by using the identifiers of a known relevant method (*method-queries*). LSI returns a list of all the methods in the software ranked by their textual similarity to the query. An advantage of this approach is that a working version of the source code is not required. However, if a program's identifiers are not meaningful, the results can be negatively affected.

For large systems, a ranked list with thousands of methods is a formidable amount of information, unless the majority of the methods that implement the feature appear near the top of the ranked list. Often, a threshold is set to limit the number of methods that users consider. The threshold may be set by a cut point, as in only the top n or only the top x percent of results are considered. The threshold can be set as at a specific value such that only the results with a similarity greater than or equal to the threshold are considered. Determining an appropriate threshold is an open research problem.

Dynamic Analysis. Using dynamic information is another approach to feature location [229, 234]. Dynamic information complements textual information since not all methods relevant to a feature may use a similar vocabulary, but they may be executed when a system is run. To collect dynamic information, an executable version of the system must be available. Users develop *scenarios* that trigger a feature. A scenario is a sequence of user inputs to a system. As scenarios are being exercised, *traces* can be collected. A trace is a list of events that occurred during the system's execution. Events can be method invocations, object instantiations, and variable accesses. This work focuses on method calls.

There are two types of traces that we consider. A *full trace* [229] captures all events from a system's start-up to shutdown. A *marked trace* [130, 192] only captures events during part of a system's execution. When the system is running, users can start and stop tracing. By starting tracing immediately before triggering a feature and stopping tracing

once the feature's behavior is observed, more of the events (methods) listed in the trace should pertain to the feature because irrelevant events are not traced.

Static Analysis. Dynamic information is only as good as the scenarios used to collect traces. If scenarios fail to invoke a feature in a certain way, relevant methods may be missing from an execution trace. Since static analysis does not rely on a program's execution, statically collected information can compensate for dynamic analysis' weaknesses [79]. Static analysis can provide a wealth of information on different types of dependencies such as control flow, data dependence, and inheritance. For this work, we use light-weight static analysis and focus on method caller-callee relationships by using a static call graph in which nodes are methods and edges represent method invocations. We obtain such a graph using JRipples[1] [35].

Additional methods relevant to a feature can be found by exploring a static call graph. Starting at a *seed method*, one that is known to be relevant to a feature, other methods pertinent to the feature can be discovered by traversing the graph. Executing a program may not invoke a relevant method, but if that relevant method has a static dependency with the seed method, static analysis can locate it. However, in the case that a method related to a feature has no static dependencies with the seed, static analysis will fail to locate relevant source code.

3.1.2 Combined Techniques

Textual, dynamic, and static information compliment each other, so in theory when working in tandem, they should produce better results than when used individually. In this work, we investigate the following combinations of analyses that produce a ranked list of results. We limit this exploratory study to techniques that rank methods to be better able to compare and evaluate the techniques.

Textual Analysis. The first feature location technique we consider employs only textual analysis, and we consider it to be our baseline approach. We evaluate two configurations of textual analysis, one using *nl-queries* as in [130] and one using our new *method-queries*. We call these approaches IR_{query} and IR_{seed}, referring to the fact that

[1] `http://jripples.sourceforge.net/`

the textual analysis used is a form of information retrieval. The IR_{query} approach was introduced by Marcus et al. [142], whereas IR_{seed} is a new version of this technique.

Textual Analysis plus Dynamic Analysis. We also examine the combination of textual and dynamic analysis for feature location. To combine these analyses, methods that are not executed are removed from the ranked list provided by textual analysis [130]. We investigate all configurations of the different types of queries and traces. Abbreviated, these configurations are $IR_{query} + Dyn_{marked}$, $IR_{query} + Dyn_{full}$, $IR_{seed} + Dyn_{marked}$, and $IR_{seed} + Dyn_{full}$, where "Dyn" stands for dynamic analysis and subscripts denote the type of trace (i.e., full or marked). $IR_{query} + Dyn_{marked}$ is like the SITIR approach [130], while $IR_{query} + Dyn_{full}$ is somewhat similar to the PROMESIR approach [160] because it uses LSI and full traces. The two other approaches are novel combinations.

Textual, Dynamic, and Static Analyses. The final feature location technique we evaluate incorporates all three types of analyses. Again, we investigate all configurations of queries and traces in conjunction with static analysis: $IR_{query} + Dyn_{marked} + Static$, $IR_{query} + Dyn_{full} + Static$, $IR_{seed} + Dyn_{full} + Static$, and $IR_{seed} + Dyn_{full} + Static$. The $IR_{query} + Dyn_{full} + Static$ approach is conceptually similar to Cerberus [62], but instead of using prune-dependency analysis, it uses light-weight static analysis. The other three combinations are new.

Unlike with combining textual and dynamic analysis, utilizing static analysis does not involve pruning an existing ranked list. Instead, static analysis entails exploring a call graph to find relevant methods and then ranking them once exploration stops. Searching begins at a seed method that is known to be relevant to the feature. The static neighbors of the seed (i.e., callers and callees) are examined to see if they meet textual and dynamic criteria. The textual criterion is a threshold similarity value, and the dynamic criterion is whether the method appears in a given trace. If the method's textual similarity is above the threshold and it was executed, it is added to the list of results, and its neighbors are added to a list of methods to be examined. Once the list of methods to examine is empty, exploration stops and the list of results is sorted by textual similarity. Cerberus [62] uses all three types of analyses. We did not use Cerberus because it does not produce a ranked list of methods and the other techniques in our evaluation do. Therefore for the sake of

comparison, we developed our own combination of textual, dynamic, and static analyses.

In total, we investigate ten different feature location techniques, many of which are novel because they involve *method-queries*. There are other possible combinations of textual, dynamic, and static analysis that we decided not to study, such as dynamic and static analysis together. We decided against including these other approaches in our study since they do not produce a ranked list and the results of using standalone versions of static and dynamic analyses are available elsewhere [39, 130, 160]. The details of how we evaluated and compared the ten approaches described above are provided in the next section.

3.2 Exploratory Study

We performed an exploratory study to evaluate the feature location techniques described in the previous section. The goal of the study was to determine which combination of analyses provides the best results and under what circumstances. This section outlines the software systems used in the study, our research goals, and the specifics on how we used each type of analysis.

3.2.1 Research Questions

We set out to seek the answers to a number of research questions in this exploratory study. These research questions (**RQ**) are as follows:

- **RQ1**: What is the best combination of textual, dynamic, and static analyses for feature location? Specifically, which techniques are most effective at finding multiple feature-relevant methods?
- **RQ2**: Which type of IR query produces better results in terms of finding multiple methods associated with a feature, an *nl-query* provided by a user (e.g. requires human effort in formulating a query) or a *method-query* using the text of a seed method (completely automatic)?
- **RQ3**: Which type of execution trace, *marked* or *full*, is better at discovering numerous methods that implement a feature?

Since this study is exploratory in nature, we did not know what to expect as the outcome, so we did not formulate any hypotheses. However, intuition and previous research results led us to conjecture that the approaches that incorporated more types of analyses would perform better than those with fewer.

3.2.2 Subject Systems

For our study, we chose two open-source Java software systems of different sizes and from different domains. jEdit[2] is a highly configurable and customizable text editor. We used version 4.3pre16, which consists of approximately 105KLOC in 910 classes and 5,530 methods. We selected four features from jEdit to study. These features were chosen from feature requests with submitted patches in the "Patches" section of the systems' online tracking software.

- **Patch #1608486, *Support for "Thick" Caret*** — Add a configurable option to make the cursor two pixels wide instead of one so it is easier to see.
- **Patch #1818140, *Edit History Text*** — Add the ability to edit the history text of searches.
- **Patch #1923613, *Reverse Regex Search*** — Add the ability to search backwards with regular expressions.
- **Patch #1849215, *Bracket Matching Enhancements*** — Add the ability to match angle brackets.

Eclipse[3], the other system in our study, is a popular integrated development environment. We used version 2.1, and it has approximately 2.3MLOC in over 7,000 classes and 89,000 methods. Like with jEdit, we selected four features from its bug tracking system. With Eclipse, we chose fixed bugs corresponding to misbehaving features. These bugs are:

- **Bug #5138**[4] – Double-click-drag to select multiple words is broken.
- **Bug #31779**[5] – UnifiedTree should ensure file/folder exists.
- **Bug #19819**[6] – Add support for Emacs-style incremental search.

[2] `http://www.jedit.org/`
[3] `http://www.eclipse.org/`
[4] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=5138`
[5] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=31779`
[6] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=19819`

- **Bug #32712**[7] – Repeated error message when deleting and file is in use.

3.2.3 Input to the Analyses

Textual Analysis. We formulated the *nl-queries* used by textual analysis by reviewing the description and comments in the thread for the patch/bug in jEdit and Eclipse's tracking systems. The *nl-queries* are listed in Table 3.1. The *method-queries* consist of the identifiers from the seed methods also listed in the table. The seed methods were randomly chosen from the patch for each feature to ensure that they do actually pertain to the feature.

Dynamic Analysis. We created one usage scenario per feature to collect traces in this study. Descriptions of the scenarios are in Table 3.1. We devised the jEdit scenarios by reading the description and comments for the patch in the bug tracking software. For Eclipse, two bug reports (#5138 and #32712) had steps to reproduce the errors, and those steps were used as the scenarios for those two features. The scenario for bug #31779 is the same as the one use in [130]. For Bug #19819, a scenario was created in which the behaviors of the incremental search feature, as described in the bug report, were exercised.

Static Analysis. The seed methods used as the starting point of static analysis are listed in Table 3.1. They were the same methods used for constructing the *method-queries* and were randomly selected from the feature's patch. As explained in Section 3.1.2, static analysis starts at the seed method and branches out in part based on a textual similarity threshold. To determine the textual similarity threshold to set for examining neighbors in a static call graph, we adapted the gap threshold technique [142, 244]. A gap threshold is found by determining the largest difference between two adjacent textual similarity values in a ranked list. The threshold is set as the larger of the two values at this location in the list. We adapted this technique to incorporate a relaxation strategy. If the size of a ranked list did not reach our minimum (e.g., ten methods), then we decreased the threshold by 0.05 and repeated the procedure again.

[7] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=32712`

Table 3.1: Queries, scenarios, and seed methods for each feature.

Feature	**Query**	**Scenario**	**Seed Method**
jEdit Patch #1608486 Support for "thick" caret.	configuration global option thick caret text area block	Start jEdit; click "Global Options" button then "Text Area;" start tracing; click "thick" checkbox then "OK;" stop tracing; exit.	EditPane.initPainter (49LOC, 114 terms)
jEdit Patch #1818140 Edit the entries in the History Text.	history text edit string menu	Start jEdit; click "Find" button; start tracing; right click in text area; select "Previously entered searches;" delete, insert, and modify an entry; click "OK;" stop tracing; exit.	ListModelEditor.createTableModel (9LOC, 18 terms)
jEdit Patch #1923613 Reverse searching with regular expressions.	reverse regex search regular expression	Start jEdit; place cursor at end of file; start tracing; click "Find" button; select "Regular Expressions" and "Backwards;" enter "[0-9]+;" click "Find" several times; stop tracing; exit.	SearchDialog.updateEnabled (30LOC, 53 terms)
jEdit Patch #1849215 Match angle brackets.	angle right find next	Start jEdit; place start tracing; cursor to right of "<" whose match is on same line; place cursor to right of "<" whose match is on another line; stop tracing; exit.	TextUtilities.findMatchingBracket (147LOC, 117 terms)

Table 3.1: (continued).

Feature	**Query**	**Scenario**	**Seed Method**
Eclipse Bug #5138 Double-click-drag to select multiple words.	mouse double click up down drag release	Start Eclipse; start tracing; click and release the mouse button; click a second time quickly and hold the mouse button down, drag and select some text; release the mouse button; stop tracing, exit.	TextViewer.mouseUp (11LOC, 36 terms)
Eclipse Bug #31779 UnifiedTree should ensure file/folder exists.	unified tree node file system folder location	Start Eclipse; start tracing; create a file from the file system in a project; refresh; stop tracing; exit.	UnifiedTree.addChildren (53 LOC, 108 terms)
Eclipse Bug #19819 Emacs-style incremental search.	incremental search	Start Eclipse; start tracing; press Ctrl+J; type search criteria; use up and down arrow keys to find matches; stop tracing; exit.	IncrementalFindAction.run (14LOC, 23 terms)
Eclipse Bug #32712 Repeated error message.	delete resource project file folder fail	Start Eclipse; create a simple project; add a file; edit foo.doc externally; start tracing; delete the project; stop tracing, exit.	ResourceTree.standardDeleteProject (78LOC, 216 terms)

3.2.4 Relevancy Assessment

Each combination of analyses is a feature location technique that produces a ranked list of methods suggested to be relevant to a feature. We restrict our evaluation to the top ten methods of each list because other researchers have shown that users are generally unlikely to look at more than ten elements on a list [157, 247]. If most of the top ten suggestions provided by a feature location approach are false positives, then the effort that would be needed to examine more results lower in the list is likely to not be worth the cost. In reviewing the top ten methods returned by each technique, there needs to be well-defined criteria for judging whether a method is relevant to a feature or not. In almost all cases, the methods that implement a feature are not documented; otherwise feature location would not be necessary. Therefore, other ways of determining a method's relevance to a feature are needed. One option is to present the top ten suggestions to an expert. If no expert is available, then if a bug related to the feature has been fixed, the methods in the patch can be used. However, the bug may only pertain to a small subset of the feature's relevant methods, so relying on a patch may give an incomplete picture of a feature's implementation. For this reason, we decided not to use this evaluation approach, even though we had patches for each feature.

An alternative is to ask programmers to identify relevant methods by exploring the source code. Robillard et al. [183] provided some guidance to participants asked to locate methods relevant to features. The participants were instructed to decide if a method was relevant by asking if it would be useful to know if the method was related to the feature if the feature had to be modified in the future. We take a similar but adapted approach in our evaluation. Instead of asking programmers to locate relevant methods on their own, we present them with lists of methods and ask them to determine the relevance of each method. In our study, the participants were provided with code and an executable, a description of a feature and how to invoke it, and the following guidelines for how to determine if a method is relevant to the feature or not.

1. Method names that are similar to the words in the feature's description are good indicators of possibly relevant code, but the method's source code should be inspected

to ensure the method is actually relevant to the feature.

2. Determine if the method is relevant to the feature by asking "*Would it be useful to know that this method is associated with the feature if I had to modify the feature in the future?*"

3. If most of the code in the method seems relevant to the feature, classify the method as *Relevant*. If some code within the method seems relevant but other code in the method is irrelevant to the feature, classify the method as *Somewhat Relevant*. If no code within the method seems relevant to the feature, classify it as *Not Relevant*.

4. If unable to classify the method by reviewing its code, explore the method's structural dependencies, i.e. what other methods call it and are called by it. If the method's dependencies seem relevant, then the method probably is also.

Having a number of programmers follow these guidelines and focusing on the agreement between the programmers eliminates any one individual's bias. We classified every method in the resulting ranked lists for all eight features without knowing which technique produced each list. To give support to the resulting categorizations, we solicited volunteers to also classify methods and compared them to ours. Four students volunteered to participate in this study. The students were enrolled in a graduate-level software engineering course. They were given ten ranked lists each containing ten methods. The ten lists corresponded to the ten different feature location techniques under evaluation. The students were not aware to which feature location technique the lists pertained. They were instructed[8] to classify the methods based on the guidelines above, and jEdit's *thick caret* feature was used. The patch for this feature has six methods, and the feature location techniques were able to find between one and three of these methods in the top ten of their ranked lists.

Figure 3.1 shows the average agreement between our classifications and the student volunteers'. To demonstrate how percent agreement was calculated, consider the following example. We classified four methods from a list of ten as relevant and six as not relevant, and a volunteer classified only three methods as relevant and seven as not relevant.

[8]See Appendix B for the instructions given to the students.

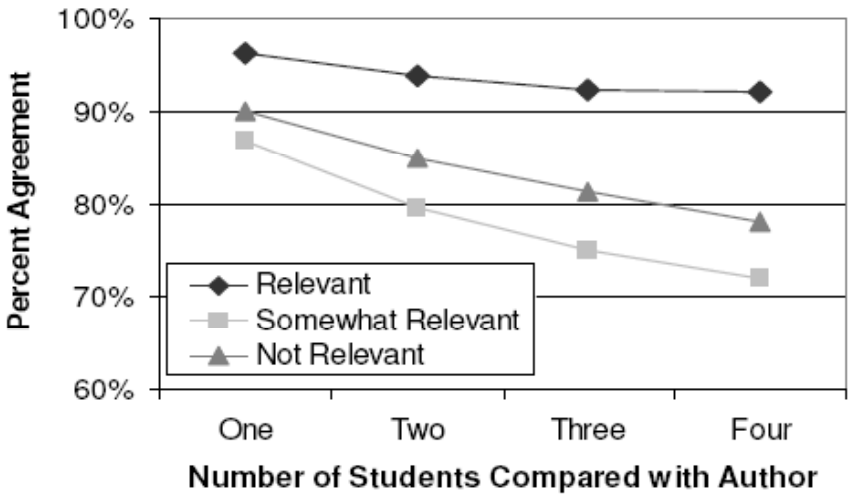


Figure 3.1: Percent agreement among the volunteers and our classifications for the jEdit thick caret feature.

The volunteer's three methods were included in the four identified by us, so the percent agreement is 90%. Nine out of ten times, both programmers agree that a method either belonged in the relevant or not relevant categories. The percent agreement was averaged over all ten lists generated by the different feature location techniques. When computing agreement between more than two programmers, all individuals involved had to categorize a method in the same way for there to be agreement. The percent agreement between us and the volunteers is high; it is always greater than 70%. The agreement declines only slightly when more individuals are taken into account. Agreement about relevant methods was highest, followed by agreement about irrelevant methods, suggesting that it is easiest to identify methods that definitely do or do not implement a feature.

The average agreement among programmers about a method's relevance in this study was higher than that observed by Robillard et al. [183]. The two approaches to evaluating method relevance differ: our study provided lists of methods for programmers to judge while Robillard et al. asked programmers to find the methods implementing a feature themselves. Also, our study allowed programmers to place methods into one of three categories to allow for uncertainty instead of a binary yes/no classification.

Table 3.2: Average percentage of the number of methods classified as relevant, somewhat relevant, and not relevant in the top ten results returned by each feature location technique for jEdit.

	Relevant	**Somewhat Relevant**	**Not Relevant**
IR_{query} [142]	12.5%	15%	72.5%
IR_{seed}	12.5%	20%	67.5%
$IR_{query} + Dyn_{marked}$ [130]	30%	20%	50%
$IR_{query} + Dyn_{full}$ [160]	15%	22.5%	62.5%
$IR_{seed} + Dyn_{marked}$	20%	15%	65%
$IR_{seed} + Dyn_{full}$	15%	27.5%	57.5%
$IR_{query} + Dyn_{marked} + Static$	30%	17.5%	52.5%
$IR_{query} + Dyn_{full} + Static$ [61]	12.5%	25%	62.5%
$IR_{seed} + Dyn_{marked} + Static$	17.5%	17.5%	65%
$IR_{seed} + Dyn_{full} + Static$	12.5%	30%	57.5%
Average	17.5%	21.25%	61.25%
Standard Deviation	7.1%	5.2%	6.9%

3.3 Results

In our study, only the top ten ranked methods returned by a feature location technique for each feature were examined. Those methods were then classified into three categories (relevant, somewhat relevant, or not relevant) as described in the previous section. The results of the jEdit and Eclipse studies are discussed in the next sections. An online appendix[9] contains the source code, classifications, and other data related to this evaluation.

3.3.1 jEdit Study Findings

The average percentage of relevant, somewhat relevant, and not relevant methods found in the top ten lists of each feature location technique are in Table 3.2 and Table 3.3. An in-depth discussion of the results is below.

RQ1. Table 3.2 lists the average number of relevant, somewhat relevant, and not relevant methods found in the top ten lists of each technique in jEdit. For jEdit, the techniques that found the most relevant methods on average were $IR_{query} + Dyn_{marked}$ and $IR_{query} + Dyn_{marked} + Static$ with 30% of the top ten methods being relevant, meaning three methods in the top ten were relevant on average. These approaches found nearly double the amount of relevant code than most of the other techniques which averaged between 12.5% and 20%. Different programmers may consider the methods classified as

[9] `http://www.cs.wm.edu/semeru/data/icpc09-feature-location/`

somewhat relevant as pertaining to the implementation of a feature, while others might not. If the somewhat relevant methods are considered important to a feature's implementation, then $IR_{query} + Dyn_{marked}$ is the best performing technique in the jEdit study with 50% of the located methods being relevant on average. At least for jEdit, the $IR_{query} + Dyn_{marked}$ feature location technique is readily able to locate many methods implementing a feature and not just a single method.

Since the $IR_{query} + Dyn_{marked}$ and $IR_{query} + Dyn_{marked} + Static$ approaches performed the same, these results suggest that adding static analysis provides no additional benefits over a combination of only textual and dynamic analysis. However, the approach that located the most relevant methods for the *edit history text* feature was $IR_{query} + Dyn_{marked} + Static$. Seventy percent of the methods in its top ten list were relevant. The methods implementing this feature have very clear structural dependencies because they can be found along the same branch of the call graph. Therefore, static analysis was easily able to identify multiple methods related to this feature. With the three other jEdit features, static analysis did not perform as expected and improve the number of relevant methods located. Incorporating static analysis yielded no more relevant methods than using a combination of textual and dynamic analysis. For jEdit's *reverse regex search* feature, a different seed than the one listed in Table 3.1 was originally selected. However, the seed method was isolated in the call graph, so static analysis could not expand far beyond it to locate more potentially relevant methods. This is one of the observed limitations of static analysis for feature location.

Another reason static analysis may not produce improved results is even when there is a dependency between a seed method and a relevant method, they may be distant from each other in the call graph. If one method along a branch in a call graph between the seed and a relevant method is not executed or has a textual similarity below the threshold, static analysis will be unable to locate the relevant method. Therefore, the ranked list is populated with other, irrelevant methods that meet both the textual and dynamic criteria when searching the call graph.

In general, combining just textual and dynamic analysis either did not affect the number of relevant methods located (*reverse regexp* feature) or slightly improved the results (*edit*

history text and *angle bracket matching* features) by pruning unexecuted methods from the ranked list. This result supports the findings of previous studies [130]. However, the combination of the two analyses did not find a substantial number of relevant methods for each feature.

For jEdit's *thick caret* feature, surprisingly, we observed that adding dynamic analysis to textual produced worse results than textual analysis alone. The StandaloneTextArea.initPainter method appears to be a code clone of EditPane.initPainter, the seed method, meant to be used when jEdit is embedded in another system. The IR_{seed} approach locates this method, but neither the $IR_{seed} + Dyn_{marked}$ nor the $IR_{seed} + Dyn_{full}$ approach can identify this method because it was not executed. This case highlights a challenge associated with using dynamic analysis for feature location. One solution is to create a better scenario, or perhaps when combining textual and dynamic analysis, if a method has a high enough textual similarity, the fact that it was not executed should be ignored.

Our goal was to locate as many methods relevant to a feature as possible. If we had set out to find only a single method to use as a starting point for searching for more methods associated with a feature, the techniques we evaluated performed with effectiveness comparable to that reported in previous studies [130, 160]. On average, at least one relevant method was found in the top ten for each feature by every technique. However, since the average number of relevant methods found by the feature location techniques is low, this work highlights the fact that finding a near-complete set of methods that implement a feature is not simple.

RQ2. Based on the jEdit data, there is no consensus on whether an *nl-query* or a *method-query* is best. For the *reverse regexp* feature, the *nl-query* performed better, while for the *thick caret* feature, the *method-query* was best. For the two other features, both queries returned the same number of relevant methods. This result suggests that using an automatically generated query of identifiers from a seed method performs just as well as a query constructed by a human, which could eliminate much of the subjectivity inherent in formulating a query.

Even though there is no clear winner, some interesting observations can still be drawn. The *nl-queries* consisted of a few words, while the *method-queries* were comprised of many

identifiers. The larger the seed methods, the more identifiers there generally were. The seed methods (refer to Table 3.1) varied in size from 9LOC and fewer than 20 identifiers (*edit history text*) to 147LOC and over 100 identifiers (*match angle brackets*). Considering only the IR_{query} and IR_{seed} results, the *method-query* for the *thick caret* feature (114 terms) performed better than the *nl-query* (8 terms) with 30% relevant vs. 10%. The wealth of identifiers in larger methods may aid textual analysis by providing more query terms, but this trend is not universal. The seed for the *angle bracket matching* feature has over 100 terms, but the two types of queries performed the same.

RQ3. On average, the use of *marked* traces produced better results than *full* traces when locating relevant methods for features in jEdit, which supports the results of previous studies as well [130]. Using *marked* traces limits the number of methods that are traced, meaning more irrelevant methods will be pruned from a ranked list. On the other hand, *full* traces were better at finding methods categorized as somewhat relevant. The methods classified as somewhat relevant generally seem to be in the call chain of relevant methods but do not directly implement the feature. We can find no explanation for why *full* traces found more somewhat relevant methods and conjecture it may be coincidental.

The nature of a feature should be considered before deciding to use *marked* traces over *full* traces. A feature like *angle bracket matching* that does not have a menu interface is suitable for *marked* traces, but for features that involve setting options in a dialog or menu, like jEdit's *thick caret* and *reverse regex* features, *full* traces might be the better option. Consider the method TextAreaOptionPane._init that adds various options for jEdit's main text area, including the *thick caret* option, to a dialog. This method was executed, but it did not appear in the *marked* trace since tracing was started after the dialog opened. *Marked* traces run the risk of omitting initialization code that *full* traces include.

3.3.2 Eclipse Study Findings

Table 3.3 lists the average number of relevant, somewhat relevant, and not relevant methods found in the top ten lists of each technique in Eclipse. Below, we discuss the results with regards to our research questions.

RQ1. For Eclipse, there were three approaches that, on average, performed the best

Table 3.3: Average percentage of the number of methods classified as relevant, somewhat relevant, and not relevant in the top ten results returned by each feature location technique for Eclipse.

	Relevant	**Somewhat Relevant**	**Not Relevant**
IR_{query} [142]	22.5%	12.5%	65%
IR_{seed}	12.5%	22.5%	65%
$IR_{query} + Dyn_{marked}$ [130]	25%	5%	70%
$IR_{query} + Dyn_{full}$ [160]	25%	12.5%	67.5%
$IR_{seed} + Dyn_{marked}$	27.5%	25%	47.5%
$IR_{seed} + Dyn_{full}$	27.5%	35%	42.5%
$IR_{query} + Dyn_{marked} + Static$	30%	12.5%	57.5%
$IR_{query} + Dyn_{full} + Static$ [61]	30%	12.5%	57.5%
$IR_{seed} + Dyn_{marked} + Static$	30%	15%	55%
$IR_{seed} + Dyn_{full} + Static$	27.5%	22.5%	50%
Average	24.75%	19.5%	55.75%
Standard Deviation	5.6%	9.6%	11.2%

at finding relevant methods: $IR_{query} + Dyn_{marked} + Static$, $IR_{query} + Dyn_{full} + Static$, and $IR_{seed} + Dyn_{marked} + Static$. Thirty percent of the top ten methods identified were relevant. When taking both relevant and somewhat relevant methods into account, the best performing approach was $IR_{seed} + Dyn_{marked} + Static$, with on average 62.5% or slightly better than six methods out of the top ten.

Unlike in jEdit, these results suggest that static analysis does aid feature location. Examining individual features, a mixed story emerges. For bugs #19819 and #32712, adding static analysis produced no improvement over a combination of textual and dynamic analysis. Bug #5138 actually saw the number of relevant methods decrease when static analysis was used. Combining textual and dynamic analysis essentially involves eliminating unexecuted methods from a ranked list, but using static analysis entails building a new list from scratch. Only methods with a static dependency to the seed are included. Therefore, methods that are located by a combined textual dynamic approach may not be found by one that uses static analysis. This is exactly what happened in the case of bug #5138. The seed method was isolated in the call graph, so static analysis was not able to branch out.

Feature location on bug #31779 resulted in the biggest improvement when adding static analysis. Ninety percent of the methods in the top ten list for $IR_{query} + Dyn_{marked} + Static$ were relevant, while 100% of the methods for $IR_{query} + Dyn_{full} + Static$ were. Static analysis was able to succeed with this feature because many of the relevant methods were located

in the same class as the seed. The results for these two approaches for this feature may have skewed Eclipse's averages. Nevertheless, this case shows that it is possible to locate near-complete feature implementations and that static analysis is a useful tool to do so.

Overall, the combination of textual and dynamic information improved results over only textual analysis, but for one feature the use of textual and dynamic information caused the number of relevant methods located to decrease. The IR_{query} technique identified JavadocDoubleClickStrategy.doubleClicked as relevant to bug #5138. However, this method is not executed in the scenario because no Javadoc comments were double clicked. Therefore, this method has no chance of being identified by an approach that uses dynamic analysis unless a new scenario is used. Alternatively, since this method has a high textual similarity, a revised combination of textual and dynamic analysis that allows for cases when a method is unexecuted but has high similarity could also solve this problem.

The purpose of this exploratory study was to learn how effective feature location techniques are at finding multiple methods relevant to a feature instead of just a single starting point. In Eclipse, all but one approach had at least 20% of its top ten located methods categorized as relevant. Most approaches found closer to 30%. These results are more encouraging and those for jEdit, but they still show room for improvement. Being able to fully locate the implementation of a feature is a difficult problem that requires further research.

RQ2. The Eclipse data showed that *method-queries* perform comparably to *nl-queries*. This outcome is similar to what was observed in jEdit. Considering the IR_{query} and IR_{seed} results for bugs #5138 and #19819, the seed methods were short (11LOC/36 terms and 14LOC/23 terms), and *nl-queries* performed better for these features. For bug #31779, the two types of queries achieved comparable results to each other, but for bug #32712 (78LOC/216 terms), the *method-query* was the winner. This possible trend of *method-queries* from longer methods performing better was also seen in jEdit, adding weight to the idea of automatically constructing queries from the identifiers of seed methods.

RQ3. In Eclipse, *marked* traces outperformed *full* traces slightly. When the same type of query was used, *marked* traces found about 5% more relevant methods than *full* traces. We attribute this outcome to *marked* traces limiting the method invocations recorded, thus

removing much noise from the resulting trace. Collecting *full* traces is difficult because they are very large and take time to collect, especially for a system like Eclipse. This fact plus the better performance of *marked* traces make them the ideal choice in most cases.

3.3.3 Discussion

Based on this exploratory study, we draw a number of notable conclusions, which are discussed below.

Method-queries* perform as well as *nl-queries. There was no clear winner when it comes to *nl-queries* vs. *method-queries*. This result is promising because it means that automatically generated queries perform just as well as ones created by humans. We observed that *method-queries* from larger seeds seem to perform the best. These results motivate further exploration into strategies for formulating queries automatically.

No feature location technique is universally successful at finding near-complete implementations of features. At best, they are good at locating a few relevant methods. This research motivates the need for feature location techniques that successfully discover as many feature-relevant methods as possible.

The effectiveness of static analysis might be tied to the effectiveness of textual analysis. The biggest difference between the results of the two systems concerns the use of static analysis. In jEdit, feature location with static analysis did not produce better results than approaches without it. In Eclipse, the best techniques used static analysis. One possible reason for this discrepancy stems from textual analysis. Static exploration of a call graph was performed using textual and dynamic criteria. If a method did not meet the textual similarity threshold, then exploration down that path of the call graph would halt. LSI generated better results for Eclipse, therefore, it is possible that static analysis was able to explore a call graph more fully and find more relevant methods in Eclipse than jEdit. Using additional types of static dependencies along with light-weight analysis may improve results.

Marked traces slightly outperform full traces. In both systems, *marked* traces were able to find slightly more relevant methods than *full* traces due to the fact that

marked traces capture a higher concentration of feature-relevant methods. However, *full* traces should be used for features that are invoked through menus.

LSI performs better on larger systems. One difference between the results of the two systems is that textual analysis yielded better results in Eclipse. There are two possible reasons for this outcome. First, Eclipse is a professional-grade system, so its naming conventions may be stricter than in jEdit, which would aid LSI. Another possible reason is that the performance of LSI has been shown to degrade on smaller corpora [62]. jEdit's corpus is small (about 7K terms and 5K methods) in comparison to Eclipse's (56K terms and 89K methods), therefore LSI's ranking strategies may be more effective with Eclipse.

The textual similarity threshold selected by the gap technique was too high. We adapted the gap threshold technique with a relaxation strategy in the case fewer than ten methods were found. The initial textual similarity selected was always too high. The relaxation strategy that we incorporated had to be used in every feature location technique involving static exploration of a call graph. In each case, the threshold had to be lowered significantly, sometimes by as much as 0.5. This observation suggests that feature-relevant methods are not always located close to each other in a call graph.

3.3.4 Threats to Validity

There are several issues that may limit the generalizations that can be drawn from our results. Foremost is the subjective manner in which the results were judged. We determined the relevance of the methods found by the feature location techniques. To minimize bias, we did not know to which approach each top ten list belonged. Also, we formalized how methods were classified by creating guidelines. For one feature, we also asked several programmers to categorize the methods and compared them to ours. Since the agreement between us and the students was high, it is reasonable to assume that the our classifications are representative of the features.

Another subjective aspect of this work is the construction of the *nl-queries* and the selection of the seed methods. To form the *nl-queries*, we used words from the change requests and bug reports. The seed methods were randomly selected from methods that

were submitted in patches to the features/bugs. Since those methods had to be changed to perform maintenance on the features, they must be relevant to the feature. However, the use of different queries and different seeds could alter the results.

Another threat to validity is that only one scenario was used to collect execution traces. Every effort was made to ensure that the scenarios dependably captured the behavior of the features, although certain aspects may have been missed. In many cases, the scenarios were based on the descriptions given in a bug report. Finally, we only studied a small number of features from two systems, both written in Java, limiting the ability to generalize our results to other types of software systems. Eclipse is a real-world system, but jEdit is rather small in comparison. This threat can be reduced if we experiment on more systems written in other languages and taken from other domains.

3.4 Related Work

Since feature location is an important part of software maintenance, there are many existing techniques. Chapter 2 gives a comprehensive overview of the research area, while this section focuses on the related work that is most pertinent to the work presented in this chapter. This section reviews these existing approaches by categorizing them as either static, dynamic, or hybrid feature location. In addition, we offer brief discussions of how our work differs from these techniques.

Most static feature location techniques are either structural or textual. Structural approaches [13, 39, 121, 179, 176] explore the relationships among classes, methods, and other program elements to locate features. We did not explore a purely structural feature location technique in this work due to the fact that the other approaches we studied ranked methods, and obtaining a ranking from only structural information is difficult. Textual approaches use comments and identifiers to locate code relevant to a feature by utilizing such techniques as information retrieval [142, 165], independent component analysis [90], and natural language processing [201]. We have focused on using IR for textual feature location. A number of tools use both structural and textual information to locate pertinent code [102, 244] by using textual information to prune irrelevant structural relationships, or

vice versa. In our work, we have not combined structural and textual techniques, but we have combined them in conjunction with dynamic analysis.

Software reconnaissance [229] is a dynamic approach to feature location that compares a trace of a program when a feature is invoked to a trace when the feature is not executed. Software reconnaissance has been recently expanded and improved [5, 77]. We did not evaluate software reconnaissance because its results are not ranked.

Hybrid feature location approaches seek to leverage the benefits provided by both static and dynamic analysis. Eisenbarth et al. [76, 118] developed a technique that is mostly dynamic and applies formal concept analysis to traces to produce a mapping of features to the program's methods. However, its results are not ranked, so this technique was not included in our exploratory study. Several approaches combine LSI and dynamic information. In PROMESIR [160], LSI is combined with SPR [5] to give a ranking of methods likely relevant to a feature. In SITIR [130], a single execution trace can be filtered using LSI to extract code relevant to the feature of interest. In this work, we have evaluated techniques similar to the PROMESIR and SITIR approaches because they represent the state of the art.

Cerberus [62] is the only approach we are aware of that combines three types of analyses for feature location. Our work is different from Cerberus as we are investigating several alternative combinations because Cerberus is not always able to locate methods relevant to some features. We also distinguish ourselves from Cerberus by examining the trade-offs of using textual, dynamic, and static analyzes for feature location and by evaluating our approaches on small and large systems.

3.5 Conclusion

This chapter presented an exploratory study evaluating the effectiveness of ten feature location approaches at finding near-complete implementations of features. Although we did not discover an approach that clearly works best in all situations, we did observe that combining analyses generally improves the results. One promising result is that *method-queries* perform comparably to a queries formed by a human. We also summarized cases

in which certain combinations of analyses were more effective than others. We used some of these observations to guide our work on developing new approaches to feature location in the next chapter.

Chapter 4

Using Data Fusion and Web Mining to Support Feature Location

Software systems are constantly changing and evolving in order to eliminate defects, improve performance or reliability, and add new functionalities. When the software engineers who maintain and evolve a system are unfamiliar with it, they must go through the program comprehension process. During this process, they obtain sufficient knowledge and understanding of at least the part of the system to which a change is to be made. An important part of the program comprehension process is *feature* or *concept location* [5, 12], which is the practice of identifying the source code that implements functionality, also known as a feature. Before software engineers can make changes to a feature, they must first find and understand its implementation.

For software developers who are unfamiliar with a system, feature location can be a laborious task if performed manually. In large software systems, there may be hundreds of classes and thousands of methods. Finding even one method that implements a feature can be extremely challenging and time consuming. Fortunately for software engineers in this situation, there are feature location techniques that automate, to a certain extent, the search for a feature's implementation.

Existing feature location techniques use different tactics to find a feature's source code.

Approaches based on information retrieval (IR) leverage the fact that identifiers and comments embed domain knowledge to locate source code that is textually similar to a query describing a feature [142]. Dynamic feature location techniques collect and analyze execution traces to identify a feature's source code based on set operations [229] or probabilistic ranking [5]. Static approaches to feature location rely on following or analyzing structural program dependencies [39, 176].

The state of the art in feature location involves integrating information from multiple sources. Researchers have recognized that combining more than one approach to feature location can produce better results than standalone techniques [62, 76, 102, 130, 160, 244]. Generally in these combined approaches, information from one source is used to filter results from another. For instance in the SITIR approach to feature location [130], a single execution trace is collected, and then IR is used to rank only the methods that appear in the trace instead of all of the system's methods. Thus, dynamic analysis is used as a filter to IR, and filtering is one way to combine information from several sources to perform feature location. Instead of using filtering, PROMESIR [160] combines the opinions of two "experts" (scenario-based probabilistic ranking [5] and IR [142]) using an affine transformation.

The idea of integrating data from multiple sources is known as *data fusion*. The sources of data have their individual benefits and limitations, but when they are combined, those drawbacks can be minimized and better results can be achieved. Data fusion is used heavily in sensor networks and geospatial applications to attain better results in terms of accuracy, completeness, or dependability. For example, the position of an object can be calculated using an inertial navigation system (INS) or global positioning system (GPS). An INS continuously calculates the position of an object with relatively little noise and centimeter-level accuracy, though over time the position data will drift and become less accurate. GPS calculates position discretely, has relatively more noise, and meter-level accuracy. However, when data from an INS and GPS are used together in the proper proportions, the GPS data can correct for the drift in the INS data. Thus the fusion of INS and GPS data produces more accurate and dependable results than if they were used separately.

Inspired by the benefits of using data fusion to integrate multiple sources of information,

this work applies data fusion to feature location. This chapter presents a data fusion model for feature location that is based on the idea that combining data from several sources in the right proportions will be effective at identifying a feature's source code. The previous chapter explored how well existing feature location techniques locate near-complete implementations of features. Since the results of the study in Chapter 3 indicated that feature location techniques are better at finding one relevant method than many, this chapter primarily focuses on finding a feature's first relevant method in a ranked list.

The data fusion model defines different types of information that can be integrated to perform feature location including textual, execution, and dependence. Textual information is analyzed by IR, execution information is collected by dynamic analysis, and dependencies are analyzed using web mining. Applying web mining to feature location is a novel idea, but it has been previously used for other program comprehension tasks, such as identifying key classes for program comprehension [239] and ranking components in a software repository [106]. Software lends itself well to web mining approaches, because like the World Wide Web, software can be represented by a graph, and that graph can be mined for useful information such as the source code that implements a feature.

This chapter makes the following contributions:

- A data fusion model for feature location is defined that integrates different types of information to locate features using IR, dynamic analysis, and web mining algorithms.
- An extensive evaluation of the feature location techniques defined in the model.
- New feature location techniques that have better effectiveness than the state of the art in feature location. Statistical analysis indicates that this improvement is significant.

In addition, all of the data used in the evaluation is made freely available online[1], and other researchers are welcome to replicate this work. Making the data available will help facilitate the creation of feature location benchmarks.

The remainder of this chapter is structured as follows. Section 4.1 introduces the data fusion model for feature location. Section 4.2 outlines the evaluation methodology and Section 4.3 discusses the results. Related work is summarized in Section 4.4, and Section 4.5 concludes.

[1] `http://www.cs.wm.edu/semeru/data/icpc10-data-fusion/`

4.1 A Data Fusion Model for Feature Location

The feature location model presented here defines several sources of information, the analyses used to derive the data, and how the information can be combined using data fusion.

4.1.1 Textual Information from Information Retrieval

Textual information in source code, represented by identifier names and internal comments, embeds domain knowledge about a software system. This information can be leveraged to locate a feature's implementation through the use of IR. Information retrieval is the methodology of searching for textual artifacts or for relevant information within artifacts. IR works by comparing a set of artifacts to a query and ranking these artifacts by their relevance to the query. There are many IR techniques that have been applied in the context of program comprehension tasks such as the Vector Space Model (VSM) [195], Latent Semantic Indexing (LSI) [59], and Latent Dirichlet Allocation (LDA) [15]. This work focuses on evaluating LSI for feature location, and the notation IR_{LSI} is used to denote that LSI is the method used to instantiate IR analysis in the model. IR_{LSI} follows five main steps [142]: creating a corpus, preprocessing, indexing, querying, and generating results.

Corpus creation. To begin the IR process, a document granularity needs to be chosen so a corpus can be formed. A document lists all the text found in a contiguous section of source code such as a method, class, or package. A corpus consists of a set of documents. For instance in this work, a corpus contains method-level granularity documents that include the text of each method in a software system.

Preprocessing. Once the corpus is created, it is preprocessed. Preprocessing involves normalizing the text of the documents. For source code, operators and programming language keywords are removed. Additionally, source code identifiers and other compound words are split (e.g., "featureLocation" becomes "feature" and "location"). Finally, stemming is performed to reduce words to their root forms (e.g., "stemmed" becomes "stem").

Index the corpus. The corpus is used to create a *term-by-document* matrix. The matrix's rows correspond to the terms in the corpus, and the columns represent documents

(i.e., source code methods). A cell $m_{i,j}$ in the matrix holds a measure of the weight or relevance of the i^{th} term in the j^{th} document. The weight can be expressed as a simple count of the number of times the term appears in the document or as a more complex measure such as term frequency-inverse-document frequency. Singular Value Decomposition (SVD) [195] is then used to reduce the dimensionality of the matrix by exploiting the co-occurrence of related terms.

Issue a query. A user formulates a natural language query consisting of words or phrases that describe the feature to be located (e.g., "print file to PDF format").

Generate the results. In the SVD model, each document corresponds to a vector. The query is also converted to a vector, and then the cosine of the angle between the two vectors is used as a measure of the similarity of the document to the query. The closer the cosine is to one, the more similar the document is to the query. A cosine similarity value is computed between the query and each document, and then the documents are sorted by their similarity values. The user inspects the ranked list, generally only reviewing the top results to decide if they are relevant to the feature.

4.1.2 Execution Information from Dynamic Analysis

Execution information is gathered via dynamic analysis, which is commonly used in program comprehension [52] and involves executing a software system under specific conditions. For feature location, these conditions involve running a test case or scenario that invokes a feature in order to collect an execution trace. For example, if the feature of interest in a text editor is *printing*, the test case or scenario would involve printing a file. Invoking the desired feature during runtime generates a feature-specific execution trace.

Most existing feature location techniques that employ dynamic analysis use it to explicitly locate a feature's implementation by analyzing patterns in traces post-mortem [5, 76, 229]. The model presented in this work takes a different approach to applying dynamic analysis for feature location. Information collected from execution traces is combined with other data sources instead of being analyzed itself. Execution information is integrated with other information by using it as a filter, as in the SITIR approach [130]

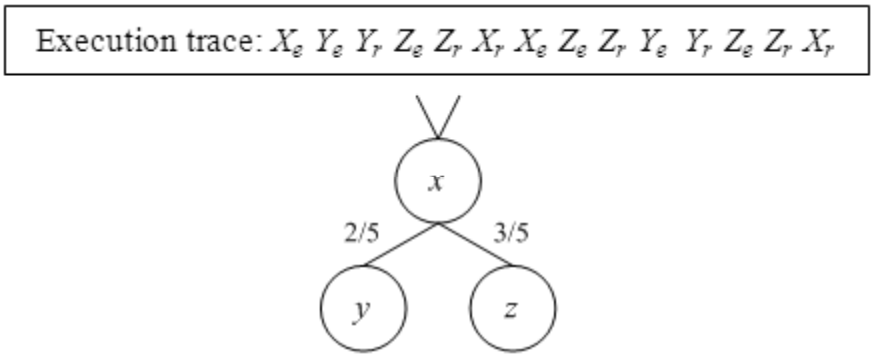


Figure 4.1: An example of an execution trace translated into a call graph with execution frequency weights on the edges. X_e is the entry to method X, and X_r is the return from method X.

where methods not executed in a feature-specific scenario are pruned from the ranked list produced by IR_{LSI}.

The model in this work takes a similar approach to using execution information (denoted as "Dyn") as a filter. By extracting information from a single trace, the sequence of method calls can be used to create a graph where nodes represent methods and edges indicate method calls. This graph is a subgraph of a static call graph that only contains methods that were executed. The edges in the graph can be weighted or weightless. When weights are used, they can be derived from execution frequency information captured by a trace. For instance, Figure 4.1 shows a portion of an execution trace where method x calls method y two times and calls method z three times. This trace is represented by a graph where the weight of the edge from x to y is 2/5, and the weight of the edge from x to z is 3/5. Alternatively, instead of normalizing the edge weights, the values on the edge from x to y can be 2, and the weight of the edge from x to z can be 3. When dynamic execution information is used in either of these ways, it is denoted with the "freq" subscript, referring to the fact that execution frequency information is used. If no weights are placed on the edges of a graph, this is denoted with the "bin" subscript, referring to the fact that only binary information about a method's execution is used.

4.1.3 Dependence Information from Web Mining

Web mining is a branch of data mining that concentrates on analyzing the structure of the World Wide Web (WWW) [49]. The structure of the WWW can be used to extract useful information. For instance, search engines use web mining to rank web pages by their relevance to a user's query. Web mining algorithms view the WWW as a graph. The graph is constructed of nodes, which represent web pages, and edges, which represent hyperlinks between pages.

Software can also be represented in graph form as a call graph. Nodes represent methods, and edges correspond to relationships or calls among methods. Therefore, web mining algorithms can be naturally applied to software to discover useful information from its structure, such as key classes for program comprehension [239], component ranks in software repositories [106], and statements that can be refined from concept bindings [127]. This work explores whether web mining can also be applied to feature location, either as a standalone technique or used as a filter to an existing feature location technique. Two web mining algorithms are discussed below.

4.1.3.1 HITS

The Hyperlinked-Induced Topic Search (HITS) [113] algorithm identifies hubs and authorities from a graph representing the WWW. Hubs are pages that have links to many other pages that contain relevant information on a topic. These pages with pertinent information are known as authorities. Good hubs point to many good authorities, and good authorities are pointed to by many hubs. Thus, hub and authority values are defined in a mutually recursive way. Let h_p stand for the hub value of page p and a_p represent the authority value of p. The hub and authority values of p are defined in Equation 4.1, where i is a page connected to p, and n is the total number of pages connected to p.

$$h_p = \sum_{i=1}^{n} a_i \text{ and } a_p = \sum_{i=1}^{n} h_i \tag{4.1}$$

To start, HITS initializes all hub and authority values to one. Then, the algorithm is run for a given number of iterations (or until the values converge), during which the hub

and authority values are updated according to Equation 4.1. The values are normalized after each iteration.

A slight variation of the HITS algorithm allows weights to be added to the links between pages. Weighted links denote relative importance. Let $w_{i \to p}$ represent the weight of the link between i and p. The formulas for hubs and authorities now become:

$$h_p = \sum_{i=1}^{n} w_{i \to p} \cdot a_i \text{ and } a_p = \sum_{i=1}^{n} w_{i \to p} \cdot h_i \tag{4.2}$$

When using software to construct a graph instead of the WWW, the nodes and edges can be determined from a static call graph or dynamic execution trace. This work concentrates on constructing a graph from execution traces. Nodes in the graph correspond to methods, and edges represent dependencies (calls) between methods. If weights are placed on the graph edges, dynamic execution frequency can be used[2]. Otherwise, if no weights are used, binary dynamic information is used. Using either frequency or binary dynamic information to construct a method call graph, the HITS algorithm can potentially be used for feature location in two ways. First, the methods in a graph can be ranked by extending the concepts of hubs and authorities to source code. Hub methods are those that call upon many other methods, while authority methods are called by a large number of other methods. Intuitively, hubs do not perform much functionality themselves but delegate to others, and authorities actually perform specific functionalities. Ranking methods in a software system by either their hub or authority values is a novel feature location technique. The notation $WM_{HITS(h,freq)}$, $WM_{HITS(h,bin)}$, $WM_{HITS(a,freq)}$, $WM_{HITS(a,bin)}$ is used, where WM refers to web mining, $HITS(h)$ and $HITS(a)$ stand for hub and authority scores respectively, and the "freq" and "bin" subscripts denote how dynamic information is used to weight the graph's edges.

The second way in which the HITS algorithm can be used for feature location is as a filter. Instead of directly using the hub and authority values to rank methods, those rankings can be combined with other information. The intuition is that the methods with high hub values will be methods that are more general purpose in nature and not specific

[2]The HITS algorithm does not require edge weights to be normalized, so the execution frequency values are used without normalization.

to a feature, i.e., methods in "god" classes. Conversely, methods with high authority values may be highly relevant to a feature. Therefore, top-ranked hub methods and bottom-ranked authority methods can be filtered from the results of other techniques such as $IR_{LSI}Dyn_{bin}$. The "top" superscript is used to represent when the top-ranked methods are filtered, and "bottom" superscript stands for the case when the bottom-ranked methods are filtered. The evaluation investigates the best method of filtering by hub and authority values.

4.1.3.2 PageRank

PageRank [31] is a web mining algorithm that estimates the relative importance of web pages. It is based on the random surfer model which states that a web surfer on any given page p will follow one of p's links with a probability of β and will jump to a random page with a probability of $(1-\beta)$. Generally, $\beta = 0.85$. Given a graph representing the WWW, let N be the total number of pages or nodes in the graph. Let $I(p)$ be the set of pages that link to p, and $O(p)$ be the pages that p links to. PageRank is defined by the equation

$$PR(p) = \frac{1-\beta}{N} + \beta \cdot \sum_{j \in I(p)} \frac{PR(j)}{|(O(j)|} \tag{4.3}$$

PageRank's definition is recursive and must be iteratively evaluated until it converges.

Like HITS, PageRank can be applied to software if a system is represented by a graph where nodes are methods executed in a trace and edges are method calls. In the PageRank algorithm, edges always have weights. When binary execution information is used, the weight of all the outgoing edges from a node is equally distributed among those edges (e.g., if x has three outgoing edges, their weight will each be 1/3). Otherwise, execution frequency information can be used for the edge weights. PageRank requires normalized values, so the execution frequency values are normalized, as in the example in Figure 4.1.

Like HITS, PageRank can be used to directly rank and locate a feature's relevant methods or as a filter to other sources of information. When used directly as a feature location technique, it is denoted as $WM_{PR(freq)}$ or $WM_{PR(bin)}$, referring to the use of frequency or binary execution information to create a graph. PageRank, applied to software, is an estimate of the global importance of a method within the system. Therefore, methods

that have global significance within a system will be ranked highly. Methods relevant to a specific feature are unlikely to have high global importance, so they may be ranked lower in the list. The evaluation examines PageRank as a feature location technique.

Since PageRank identifies methods of global importance, instead of using it as a standalone feature location technique, it can be used as a filter to be combined with other sources of information. Pruning the top-ranked PageRank methods from consideration may produce better feature location results. The "top" and "bottom" superscripts denote that the top and bottom results returned by PageRank are filtered. The evaluation explores the best way to use PageRank as a filter.

4.1.4 Fusions

Data fusion combines information from multiple sources to achieve potentially more accurate results. For feature location, this model has defined three information sources derived from three types of analysis: information retrieval, execution tracing, and web mining. This subsection outlines the feature location techniques instantiated within the model that are evaluated. Table 4.1 lists all of the techniques.

Information Retrieval via LSI. This feature location technique, introduced in [142], ranks all methods in a software system based on their relevance to a query. Only one source of information is used, so no data fusion is performed. This approach is referred to as IR_{LSI}.

Information Retrieval and Execution Information. The idea of fusing IR with dynamic analysis is used by the SITIR approach [130] and is the state of the art of feature location techniques that rank program elements (e.g., methods) by their relevance to a feature. A single feature-specific execution trace is collected. Then, LSI ranks all the methods in the trace instead of all the methods in the system. Thus dynamic information is used as a filter to eliminate methods that were not executed and therefore are less likely to be relevant to the feature. In this work, this technique is abbreviated $IR_{LSI}Dyn_{bin}$ and represents the baseline for comparison. Note that the $IR_{LSI}Dyn_{freq}$ approach is not evaluated. It filters the same methods as $IR_{LSI}Dyn_{bin}$ because it only matters whether a method was executed or not.

Table 4.1: The feature location techniques evaluated.

IR & Dynamic Analysis	IR_{LSI}
	$IR_{LSI}Dyn_{bin}$
Web Mining	$WM_{HITS(h,bin)}$
	$WM_{HITS(h,freq)}$
	$WM_{HITS(a,bin)}$
	$WM_{HITS(a,freq)}$
	$WM_{PR(bin)}$
	$WM_{PR(freq)}$
IR, Dyn, & HITS	$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{top}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{bottom}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{top}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{bottom}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{top}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{bottom}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{top}$
	$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{bottom}$
IR, Dyn, & PageRank	$IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{top}$
	$IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{bottom}$
	$IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{top}$
	$IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{bottom}$

Web Mining. The HITS and PageRank algorithms can be used as feature location techniques that rank all methods in an execution trace using either binary or frequency information. Web mining has not been applied to feature location before; therefore all of the approaches involving web mining are novel. Table 4.1 lists all the feature location techniques based on web mining.

Information Retrieval, Execution Information, and Web Mining. Applying data fusion, IR, execution tracing, and web mining can be combined to perform feature location. This work proposes the use of web mining as a filter for $IR_{LSI}Dyn_{bin}$'s results in order to eliminate methods that are irrelevant. Figure 4.2 illustrates the process. Each web mining algorithm can be applied to binary or execution frequency information. To combine $IR_{LSI}Dyn_{bin}$ and web mining, the top or bottom web mining results can be pruned from $IR_{LSI}Dyn_{bin}$'s ranked list. If the results returned by a standalone web mining technique rank methods that are relevant to a feature at the top of the list, then methods at the bottom of the list can be filtered from consideration. However, since the standalone web mining techniques are based on a dynamically-constructed call graph, the resulting rankings could

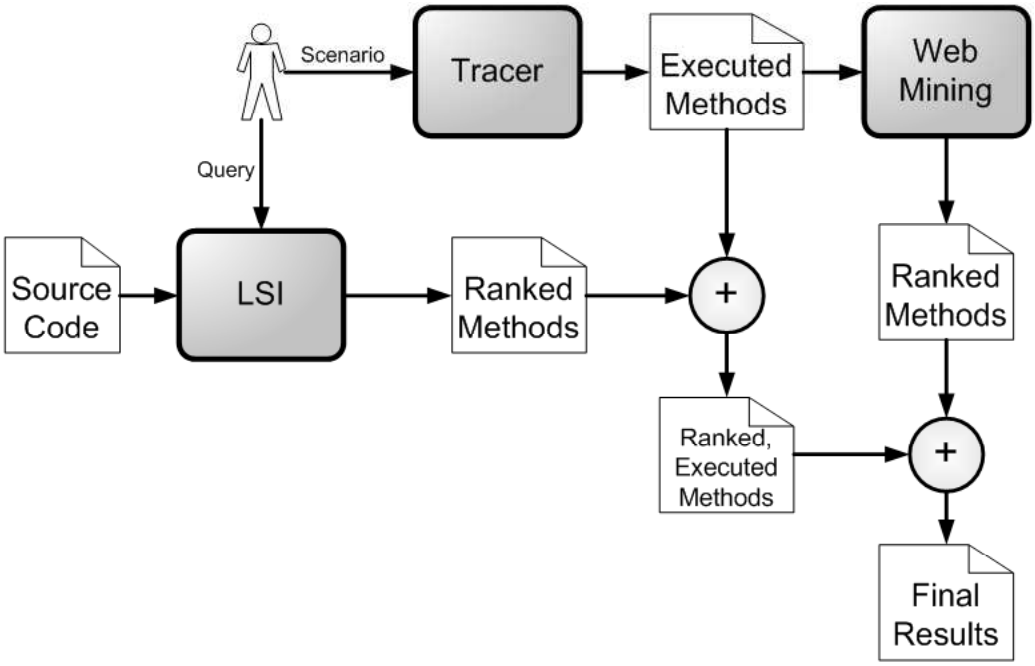


Figure 4.2: Combining textual analysis, dynamic analysis, and web mining for feature location.

be similar across many different features, meaning the top-ranked results are not relevant to the feature. In this case, those top-ranked results are eliminated from consideration. For example, $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{top}$ is a feature location technique that uses IR to rank all of the executed methods by their relevance to a query. A graph is constructed using binary execution information from a trace, and the methods in the graph are ranked according to their HITS hub values. Finally, the top methods from the HITS hub rankings are pruned from the $IR_{LSI}Dyn_{bin}$ results. In this technique, methods with high HITS hub values are filtered. Table 4.1 lists all of the feature location techniques that filter $IR_{LSI}Dyn_{bin}$'s results using HITS or PageRank.

4.2 Experimental Evaluation

This section describes the design of a case study to assess the feature location techniques defined by the data fusion model. The evaluation seeks to answer the following research questions:

RQ1: Does combining web mining algorithms with an existing approach to feature

location improve its effectiveness?

RQ2: Which web-mining algorithm, HITS or PageRank, produces better results?

The answers to these research questions will help reveal the best instantiation of the data fusion model.

4.2.1 Systems and Benchmarks

The evaluation was conducted on two open source Java software systems: Eclipse and Rhino. Eclipse[3] is an integrated development environment. Version 3.0 has approximately 10K classes, 120K methods, and 1.6 million lines of code. Forty-five features from Eclipse were studied. The features are represented by bug reports submitted to Eclipse's online issue tracking system[4]. The bug reports are change requests that pertain to faulty features. The bug reports provide steps to reproduce the problem, and these steps were used as scenarios to collect execution traces. Table 4.2 lists information about the size of the collected traces. The short descriptions in the bug reports were used as the IR queries. The bug reports also have submitted patches that detail the code that was changed to fix the bug. The modified methods are considered to be the "gold set" of methods that implement the feature. Since their code had to be altered to correct a problem with the feature, they are likely to be relevant to the feature. These gold set methods are used as the benchmark to evaluate the feature location techniques. This way of determining a feature's relevant methods from patches has also been used by other researchers [130, 132, 160].

The other system evaluated is Rhino, a Java implementation of JavaScript. Rhino[5] version 1.5 consists of 138 classes, 1,870 methods, and 32,134 lines of code. Rhino implements the ECMAScript specification[6]. The Rhino distribution comes with a test suite, and individual test cases in the suite are labeled with the section of the specification they test. Therefore, these test cases were used to collect execution traces for 241 features. The text from the corresponding section of the specification was used to formulate IR queries. For the gold set benchmarks for each feature, the mappings of source code to features

[3] `http://www.eclipse.org/`
[4] `https://bugs.eclipse.org/`
[5] `http://www.mozilla.org/rhino/`
[6] `http://www.ecmascript.org/`

Table 4.2: Descriptive statistics on the execution traces. The columns represent the minimum, maximum, lower quartile, median, upper quartile, mean, and standard deviation. Forty-five traces were collected for Eclipse, and 241 for Rhino.

		Min	**Max**	**25%**	**Med**	**75%**	$\boldsymbol{\alpha}$	$\boldsymbol{\mu}$
Eclipse	Methods	88K	1.5MM	312K	525K	1MM	666K	406K
	Unique	1.9K	9.3K	3.9K	5K	6.3K	5.1K	2K
	Size-MB	9.5	290	55	98	202	124	83
	Nesting*	22	178	37	54	71	59	32
	Threads	1	26	7	10	12	10	5
Rhino	Methods	160K	12MM	612K	909K	1.8MM	1.8MM	2.3MM
	Unique	777	1.1K	870	917	943	912	54
	Size-MB	18	1,668	71	104	214	210	273
	Nesting*	25	37	28	27	28	28	1
	Threads	1	1	1	1	1	1	0

* Nesting is based on the average nesting level per feature.

made available by Eaddy et al. [63] were used. They considered the sections of the ECMAScript documentation to be features and associated code with each following the prune dependency rule which states: "A program element is relevant to a [feature] if it should be removed, or otherwise altered, when the [feature] is pruned" [62]. Their mappings are made publically available online[7] and have been used in several other research evaluations [62, 63].

The position of the first relevant method from the gold set was used as the primary means to evaluate the feature location techniques and is referred to as the *effectiveness measure* [160]. Techniques that rank relevant methods near the top of the list are more effective because they reduce the number of false positives a developer has to consider. The effectiveness measure is an accepted metric to evaluate feature location techniques. It is used here instead of precision and recall to be consistent with previous approaches [130, 160] and because feature location techniques have been shown to be better at finding one relevant method for a feature as opposed to many [173]. However, the evaluation also investigates how well the techniques locate all of a feature's relevant methods.

4.2.2 Hypotheses

Several null hypotheses were formed to test whether the performance of the baseline feature location technique improves with the use of web mining. The testing of the hypotheses is

[7] `http://www.cs.columbia.edu/~eaddy/concerntagger/`

based on the effectiveness measure. Two null hypotheses are presented here; the other hypotheses can be derived analogously.

$\boldsymbol{H_{0,WM_{PR(bin)}}}$: There is no significant difference between the effectiveness of $WM_{PR(bin)}$ and the baseline ($IR_{LSI}Dyn_{bin}$).

$\boldsymbol{H_{0,IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}}}$: There is no significant difference between the effectiveness of $IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}$ and the baseline ($IR_{LSI}Dyn_{bin}$).

If a null hypothesis can be rejected with high confidence, an alternative hypothesis that states that a technique has a positive effect on the ranking of the first relevant method can be supported. The corresponding alternative hypotheses to the null hypotheses above are given. The remaining alternative hypotheses are formulated in a similar manner.

$\boldsymbol{H_{A,WM_{PR(bin)}}}$: The effectiveness of $WM_{PR(bin)}$ is significantly better than the baseline.

$\boldsymbol{H_{A,IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}}}$: The effectiveness of $IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}$ is significantly better than the baseline.

4.2.3 Data Collection and Analysis

The primary data collected in the evaluation is the effectiveness measure. For each feature location technique, there are 45 data points for Eclipse and 241 for Rhino, one for each feature. Descriptive statistics of the effectiveness measure for each system are reported that summarize the data in terms of mean, median, minimum, maximum, lower quartile, and upper quartile.

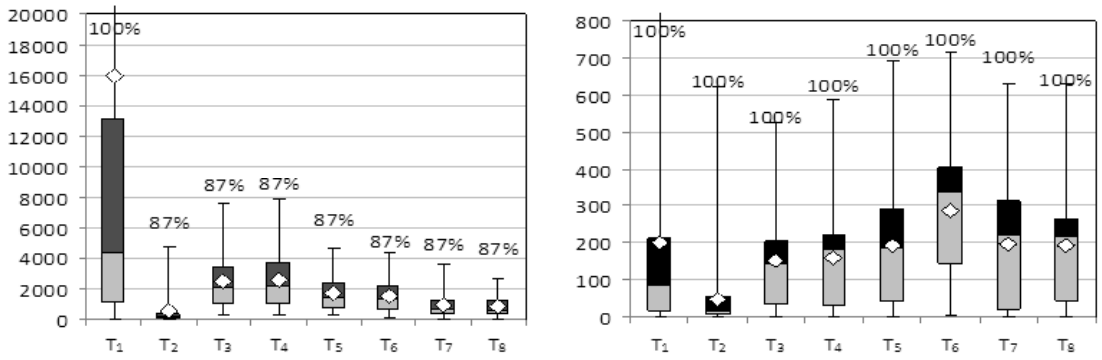
The feature location techniques can also be evaluated by how many features for which they can return at least one relevant result. Many of the techniques in the model filter methods from consideration, and some of those methods may belong to the gold set. It is possible for a technique to filter out all of a feature's gold set methods and return no relevant results. Therefore, the percentage of features for which a technique can locate at least one relevant method is reported.

If a feature location technique ranks one of a feature's relevant methods higher than another technique, then the first approach is more effective. Every feature location technique can be compared to every other technique in this manner, and the percentage of times the first technique is more effective is reported.

Data on whether one technique is more effective than another is not enough. Statistical analysis must be performed to determine if the difference between the effectiveness of two techniques is significant. The Wilcoxon Rank Sum test [48] is used to test if the difference between the effectiveness measures of two feature location techniques is statistically significant. Essentially, the test determines if the decrease in the number of false positives reported by one technique as compared to another is significant. The Wilcoxon test is a non-parametric test that accepts paired data. Since a technique may not rank any of a feature's gold set methods, it would have no data to be paired with the data from another feature location technique. Therefore, only cases where both techniques rank a method are input to the test. In this evaluation, the significance level of the Wilcoxon Rank Sum test is $\alpha = 0.05$.

4.3 Results and Discussion

This section presents the results of using the feature location techniques listed in Table 4.1 to identify the first relevant method of 45 features of Eclipse and 241 features of Rhino. Figure 4.3 and Figure 4.4 show box plots representing the descriptive statistics of the effectiveness measure for Eclipse and Rhino. The y-axis represents the effectiveness measure. The graphs for Eclipse and Rhino have different scales because Eclipse has more methods. Figure 4.3 plots the feature location techniques based on IR (T_1), IR and dynamic analysis (T_2), and web mining as a standalone approach (T_3 through T_8). Figure 4.4 shows the techniques that combine IR, dynamic analysis, and web mining (T_2 through T_{13}). $IR_{LSI}Dyn_{bin}$ is also included in this figure for reference since it represents the baseline for comparison. In Figure 4.3 and Figure 4.4, the diamonds represent the average effectiveness measure. The dark grey and light grey boxes stand for the upper and lower quartiles, respectively, and the line between the boxes represents the median. The whiskers above and below the boxes denote the minimum and maximum effectiveness measure. In some cases, the maximum is beyond the scale of the graphs. The figures also report for each feature location technique, the percentage of features for which the technique was able to identify at least one relevant method.



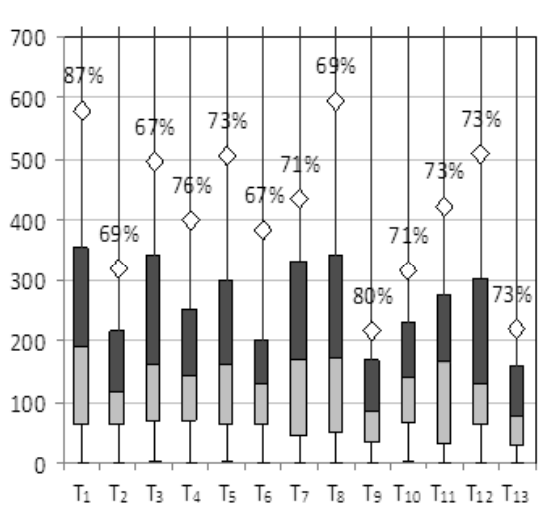
(a) Eclipse. (b) Rhino.

$\mathbf{T_1} : IR_{LSI}$ $\mathbf{T_5} : WM_{HITS(a,freq)}$
$\mathbf{T_2} : IR_{LSI}Dyn_{bin}$ $\mathbf{T_6} : WM_{HITS(a,bin)}$
$\mathbf{T_3} : WM_{PR(freq)}$ $\mathbf{T_7} : WM_{HITS(h,freq)}$
$\mathbf{T_4} : WM_{PR(bin)}$ $\mathbf{T_8} : WM_{HITS(h,bin)}$

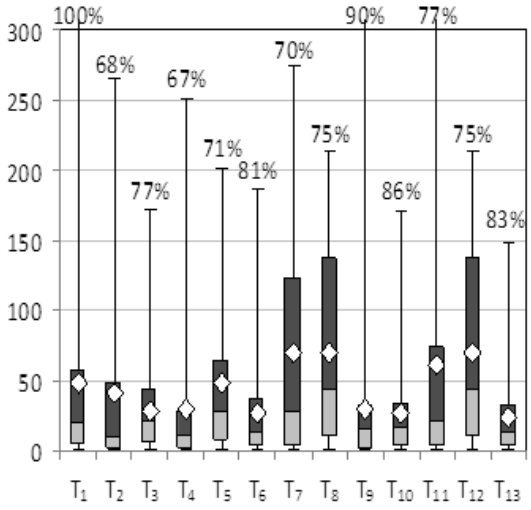
Figure 4.3: The effectiveness measure for the standalone web mining feature location techniques applied to 45 features in Eclipse and 241 features in Rhino. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

The box plots in Figure 4.3 show that using web mining as a standalone feature location technique produces results that are comparable to IR_{LSI} even though no query is used. However, these techniques are less effective than the state of the art, no matter the web mining algorithm used. Feature location based on PageRank, HITS hub values, or HITS authority values has higher effectiveness than $IR_{LSI}Dyn_{bin}$. PageRank's effectiveness was the lowest, followed by HITS authorities and HITS hubs. Overall, there is little difference between the use of binary and execution frequency information. It is surprising that ranking methods by their hub values is more effective than ranking them by their authority values. Intuitively, hubs are methods that delegate functionality to authorities which actually implement it. Therefore, authorities should be more valuable for feature location, but this was not observed.

Even though feature location techniques based on standalone web mining are not more effective than the state of the art, when web mining is used as a filter to IR, the results significantly improve in some cases. Figure 4.4 presents box plots of the effectiveness measure of the techniques that used web mining to filter $IR_{LSI}Dyn_{bin}$'s results. The filters prune either the top or bottom methods ranked by a web mining algorithm. The



(a) Eclipse.



(b) Rhino.

$\mathbf{T_1}$: $IR_{LSI}Dyn_{bin}$
$\mathbf{T_2}$: $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{top[40,60]\%}$
$\mathbf{T_3}$: $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{bottom[20,70]\%}$
$\mathbf{T_4}$: $IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{top[40,60]\%}$
$\mathbf{T_5}$: $IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{bottom[10,70]\%}$
$\mathbf{T_6}$: $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{top[30,70]\%}$
$\mathbf{T_7}$: $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{bottom[40,60]\%}$
$\mathbf{T_8}$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{top[10,70]\%}$
$\mathbf{T_9}$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{bottom[60,50]\%}$
$\mathbf{T_{10}}$: $IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{top[20,70]\%}$
$\mathbf{T_{11}}$: $IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{bottom[40,40]\%}$
$\mathbf{T_{12}}$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{top[10,70\%]}$
$\mathbf{T_{13}}$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{bottom[70,60]\%}$

Figure 4.4: The effectiveness measure for the feature location techniques that use web mining as a filter. The top and bottom percentages in brackets have two values. The first value is the percentage used in Eclipse, and the second is the percentage used in Rhino. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

threshold for the percent of methods to filter was selected for each technique individually such that at least one gold set method remained in the results for 66% of the features. In Eclipse, $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ had the best effectiveness measure on average. In Rhino, $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ was the most effective technique. In fact, all of the techniques that use web mining to filter IR are more effective than $IR_{LSI}Dyn_{bin}$ in Eclipse by 13% to 62% on average. In Rhino, most of the IR plus web mining techniques have an average effectiveness 1% to 51% better than $IR_{LSI}Dyn_{bin}$ except for $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{bottom}$, $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{top}$, $IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{bottom}$, and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{top}$. These results help answer **RQ1** because they lend strong support to the fact that integrating the ranking of methods using web mining with information retrieval is a very effective way to perform feature location. In regards to **RQ2**, the techniques based on HITS were generally more effective than the PageRank approaches, so HITS, used either as a standalone technique or as a filter, seems better suited to the task of feature location.

In addition to measuring the effectiveness of each of the feature location techniques, the new approaches based on web mining were directly compared to IR_{LSI} and $IR_{LSI}Dyn_{bin}$. Table 4.3 shows for each new technique, the percent of times it ranks a method from a feature's gold set lower than the existing approaches. The table shows a different view of the data presented in Figures 4.3 and 4.4. It shows on a case-by-case basis, which feature location technique is more effective. The data in this table is derived from the subset of methods that are ranked by both techniques, while Figures 4.3 and 4.4 show data for all methods. In Table 4.3, if one approach ranks a method and another does not, the method is not included in the reported data. The table shows that feature location techniques based solely on web mining never have better effectiveness than $IR_{LSI}Dyn_{bin}$. On the other hand, the techniques that use web mining as a filter routinely rank methods higher than $IR_{LSI}Dyn_{bin}$. This finding also helps answer **RQ1**: combining web mining with existing approaches improves their effectiveness. **RQ2** addresses which of the two web mining algorithms is more effective. Based on the results in Table 4.3, the techniques based on HITS are more effective than the PageRank techniques.

Table 4.3: For each feature location technique listed in a row, the percentage of times its effectiveness measure is better than the technique in the corresponding column is given.

	Eclipse		**Rhino**	
	$\boldsymbol{IR_{LSI}}$	$\boldsymbol{IR_{LSI}Dyn_{bin}}$	$\boldsymbol{IR_{LSI}}$	$\boldsymbol{IR_{LSI}Dyn_{bin}}$
$IR_{LSI}Dyn_{bin}$	97%	X	91%	X
$WM_{PR(freq)}$	59%	13%	49%	20%
$WM_{PR(bin)}$	59%	10%	44%	19%
$WM_{HITS(a,freq)}$	67%	18%	45%	15%
$WM_{HITS(a,bin)}$	56%	18%	25%	6%
$WM_{HITS(h,freq)}$	77%	26%	45%	20%
$WM_{HITS(h,bin)}$	77%	26%	41%	22%
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{top}$	97%	90%	85%	72%
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{bottom}$	100%	83%	83%	63%
$IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{top}$	97%	91%	85%	73%
$IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{bottom}$	97%	94%	82%	54%
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{top}$	97%	90%	88%	74%
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{bottom}$	97%	94%	82%	53%
$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{top}$	97%	94%	72%	40%
$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{bottom}$	97%	97%	93%	88%
$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{top}$	97%	94%	85%	68%
$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{bottom}$	97%	91%	73%	60%
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{top}$	97%	94%	72%	40%
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{bottom}$	97%	97%	89%	81%

4.3.1 Statistical Analysis

The Wilcoxon Rank Sum test was used to test if the difference between the effectiveness measures of two feature location techniques is statistically significant. Table 4.4 shows the results of the test (p-values) for all of the techniques based on web mining as compared to $IR_{LSI}Dyn_{bin}$ and if the null hypotheses can be rejected based on the p-values. In the table, statistically significant results are presented in boldface. None of the approaches in which web mining is used as a standalone technique have statistically significant results. However in Eclipse, all of the feature location techniques that employ web mining as a filter to IR have significantly better effectiveness than $IR_{LSI}Dyn_{bin}$. Likewise in Rhino, most of the approaches that use web mining as a filter have statistically significant results with a few exceptions. Therefore, the null hypotheses for these approaches that do not have significant results for both systems cannot be rejected. However, for the techniques with statistically significant results for both Eclipse and Rhino, their null hypotheses are rejected, and there is evidence to suggest that the corresponding alternative hypotheses can be supported. These feature location techniques have better effectiveness than the baseline

Table 4.4: The results of the Wilcoxon test.

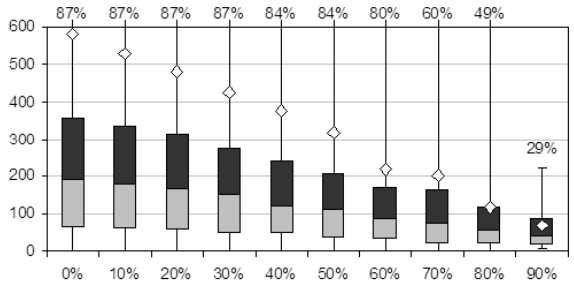
	Eclipse	Rhino	Null Hypothesis
$WM_{PR(freq)}$	1	1	Not Rejected
$WM_{PR(bin)}$	1	1	Not Rejected
$WM_{HITS(a,freq)}$	1	1	Not Rejected
$WM_{HITS(a,bin)}$	1	1	Not Rejected
$WM_{HITS(h,freq)}$	1	1	Not Rejected
$WM_{HITS(h,bin)}$	1	1	Not Rejected
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{top}$	**< 0.0001**	**< 0.0001**	**Rejected**
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{bottom}$	**0.004**	**0**	**Rejected**
$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}$	**< 0.0001**	**< 0.0001**	**Rejected**
$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{bottom}$	**< 0.0001**	0.74	Not Rejected
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{top}$	**0**	**< 0.0001**	**Rejected**
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{bottom}$	**< 0.0001**	0.99	Not Rejected
$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{top}$	**0**	1	Not Rejected
$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$	**< 0.0001**	**< 0.0001**	**Rejected**
$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{top}$	**< 0.0001**	**< 0.0001**	**Rejected**
$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{bottom}$	**< 0.0001**	1	Not Rejected
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{top}$	**0**	1	Not Rejected
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$	**< 0.0001**	**< 0.0001**	**Rejected**

technique.

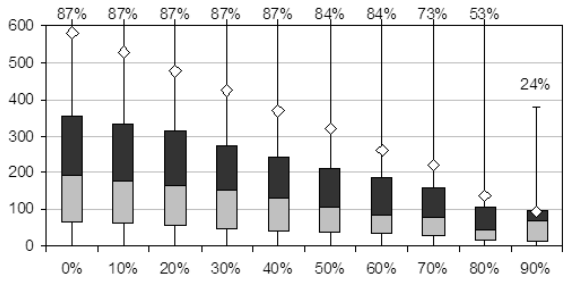
4.3.2 Impact of the Selection of a Threshold

The results in the previous section for the techniques that use web mining as a filter present only one possible threshold for what percentage of the top or bottom web mining results to eliminate from the baseline results. The threshold was chosen such that a given feature location technique returned at least one relevant method for at least 66% of the features studied. This section examines how varying the filtering threshold impacts the results, focusing on the techniques with the lowest average effectiveness, $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$. Figure 4.5 shows, for Eclipse, box plots of the average effectiveness of the two techniques with different filtering thresholds. Figure 4.6 shows the results for Rhino.

Not surprisingly, the higher the filtering threshold, the lower the average effectiveness since more methods are eliminated from consideration. However, there is a tradeoff; the improvement in effectiveness comes at the cost of completeness. The values above the boxes in Figures 4.5 and 4.6 represent the percentage of features for which the technique

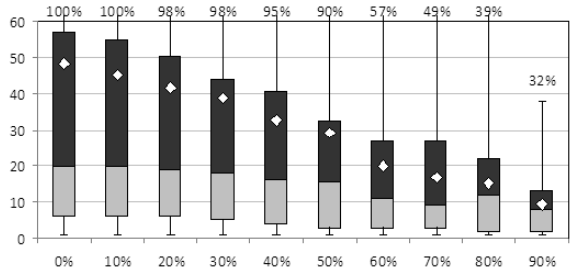


(a) $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$

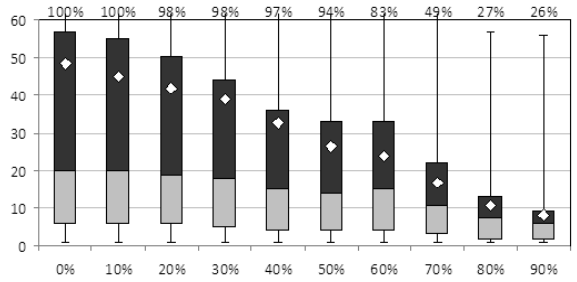


(b) $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$

Figure 4.5: Summary of the effectiveness measure of $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ at different filtering thresholds for the 45 features of Eclipse. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



(a) $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$



(b) $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$

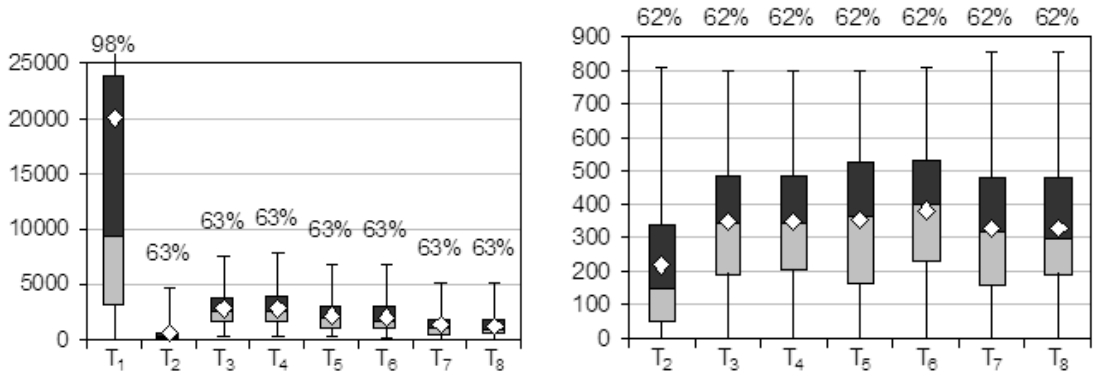
Figure 4.6: Summary of the effectiveness measure of $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ at different filtering thresholds for the 241 features of Rhino. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

was able to locate at least one relevant method. When a higher percentage of the HITS hubs results are filtered, the techniques find at least one relevant method for fewer features. For instance in Eclipse with $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$, when the bottom 90% of the HITS hubs results are pruned from the baseline, the average effectiveness is 67, but the technique can identify a relevant method for only 29% of features. Setting the threshold too high means methods that are relevant to a feature are considered false negatives and removed from the results. Therefore at least with the $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ techniques, a threshold of of 50%–60% yields acceptable results. Selecting appropriate thresholds for individual features remains part of our future work.

4.3.3 Locating All of a Feature's Methods

Chapter 3 explored how effective existing feature location techniques are at finding near-complete implementations of features and found that the existing techniques showed room for improvement. So far, this chapter has focused on the effectiveness of feature location only in terms of the position of the first relevant method (i.e., the effectiveness measure). However, since gold sets defining all the methods relevant to a feature were available, the feature location techniques can also be evaluated in terms of how well they locate all of a feature's methods. Figures 4.7 and 4.8 show box plots summarizing the average position of all of a feature's relevant methods. Figure 4.7 presents the results for IR_{LSI}, $IR_{LSI}Dyn_{bin}$, and the standalone web mining feature location techniques, while Figure 4.8 shows the results for the baseline and the techniques that use web mining as a filter.

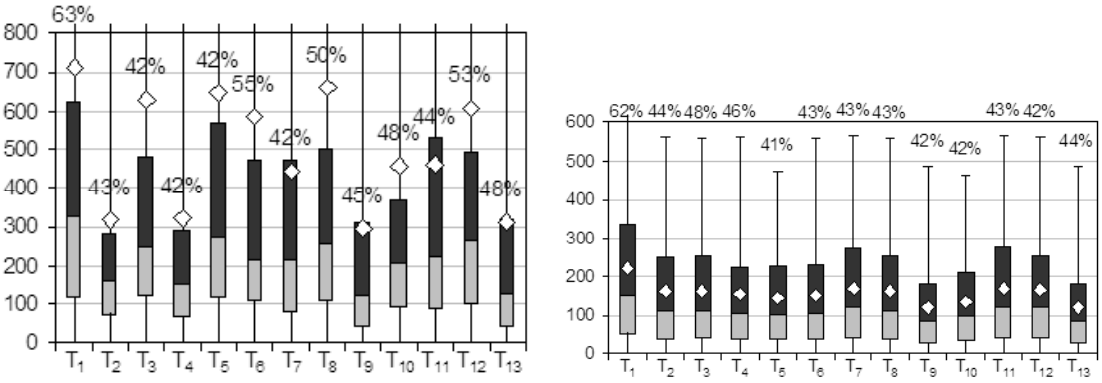
Figure 4.7 shows that the baseline approach, $IR_{LSI}Dyn_{bin}$ is the more effective at locating all of a feature's relevant methods than the standalone web mining techniques. However, using web mining as a filter improves the average effectiveness of locating all of the methods from a feature's gold set, as seen in Figure 4.8. As with the effectiveness measure results presented earlier, $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ were the two most effective techniques.



(a) Eclipse. (b) Rhino.

$\mathbf{T_1} : IR_{LSI}$
$\mathbf{T_2} : IR_{LSI}Dyn_{bin}$
$\mathbf{T_3} : WM_{PR(freq)}$
$\mathbf{T_4} : WM_{PR(bin)}$
$\mathbf{T_5} : WM_{HITS(a,freq)}$
$\mathbf{T_6} : WM_{HITS(a,bin)}$
$\mathbf{T_7} : WM_{HITS(h,freq)}$
$\mathbf{T_8} : WM_{HITS(h,bin)}$

Figure 4.7: The average position of all gold set methods for the standalone web mining feature location techniques applied to 45 features of Eclipse and 241 features of Rhino. The values above the boxes represent the percent of all the gold set methods the technique could locate.



(a) Eclipse. (b) Rhino.

$\mathbf{T_1} : IR_{LSI}Dyn_{bin}$
$\mathbf{T_2} : IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{top[50,30]\%}$
$\mathbf{T_3} : IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{bottom[20,30]\%}$
$\mathbf{T_4} : IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{top[50,30]\%}$
$\mathbf{T_5} : IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{bottom[20,40]\%}$
$\mathbf{T_6} : IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{top[20,30]\%}$
$\mathbf{T_7} : IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{bottom[40,30]\%}$
$\mathbf{T_8} : IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{top[10,30]\%}$
$\mathbf{T_9} : IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{bottom[60,40]\%}$
$\mathbf{T_{10}} : IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{top[20,40]\%}$
$\mathbf{T_{11}} : IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}^{bottom[40,30]\%}$
$\mathbf{T_{12}} : IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{top[10,30\%]}$
$\mathbf{T_{13}} : IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{bottom[60,40]\%}$

Figure 4.8: The average position of all gold set methods for the feature location techniques that use web mining as a filter applied to 45 features of Eclipse and 241 features of Rhino. The top and bottom percentages in brackets have two values. The first value is the percentage used in Eclipse, and the second is the percentage used in Rhino. The values above the boxes represent the percent of all the gold set methods the technique could locate.

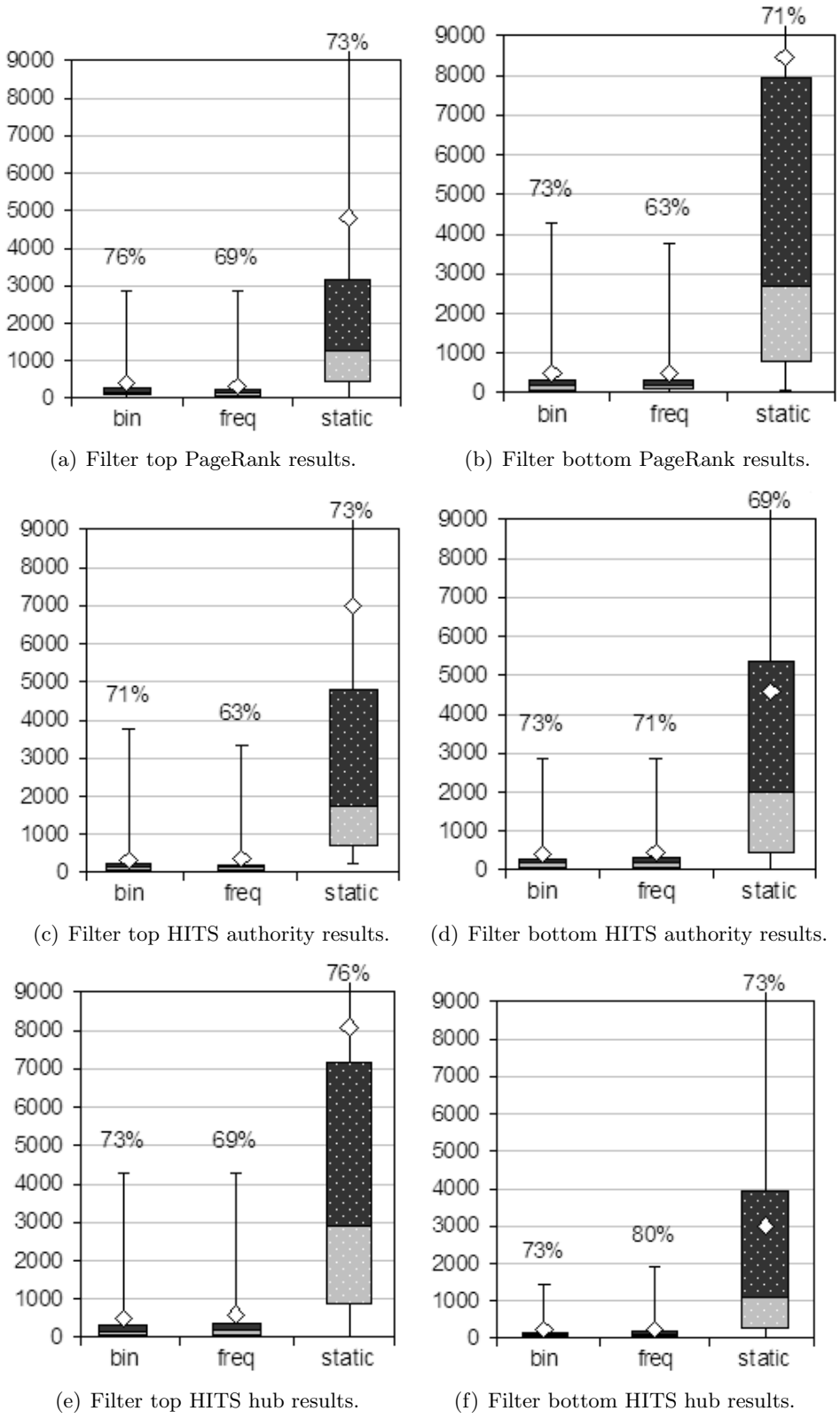
4.3.4 Using a Static Call Graph

All of the feature location techniques investigated have leveraged a call graph that is constructed from execution traces specific to each feature. Collecting execution traces is computationally expensive and time consuming. This section explores whether comparable results can be achieved using a static call graph. The tradeoff is that only one static call graph is needed instead of a different dynamic call graph for each feature, but a static call graph is generalized and not feature-specific.

Figure 4.9 shows, for Eclipse, summaries of the effectiveness measure for each of the feature location techniques based on using web mining as a filter. Figure 4.10 shows the results for Rhino. In each graph, the first plot represents using a dynamic call graph with binary weights and the second corresponds to using a dynamic call graph with execution frequency weights. The third, patterned plot represents using a static call graph. For example, Figure 4.9(a) compares the results of $IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}$, $IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{top}$, and filtering PageRank's top-ranked methods from a static call graph from $IR_{LSI}Dyn_{bin}$'s results.

Figure 4.9 shows that in Eclipse, using a static call graph is not as effective as using a dynamically-constructed call graph. A static call graph includes all of a system's methods, not just those that were executed. Eclipse has over 84,000 methods, so using a static call graph significantly increases the number of methods that need to be ranked. This increase in the number of methods leads to a decrease in effectiveness because there are more false positives in the ranked list.

Figure 4.10 shows the results of using a dynamic call graph and a static call graph for each feature location technique that uses web mining as a filter in Rhino. Unlike the Eclipse results, using a static call graph in Rhino has comparable effectiveness. In general, the static approaches are not quite as effective as the dynamic ones, but the difference is not large. In Rhino, using a static call graph gives results that are close to those when using a dynamic call graph without the cost of collecting traces. Rhino is a smaller system than Eclipse, so ranking all of its methods instead of only those that were executed introduces fewer false positives. There may be other factors in why the static results are comparable



(a) Filter top PageRank results.

(b) Filter bottom PageRank results.

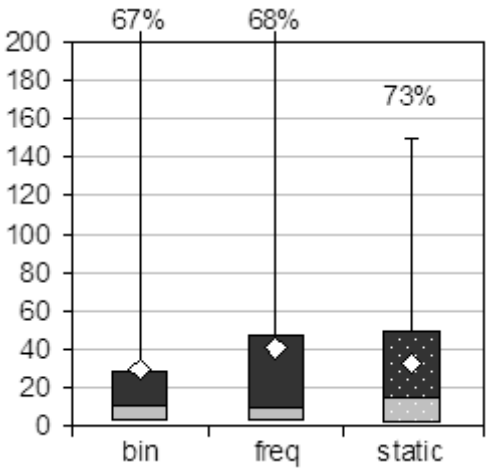
(c) Filter top HITS authority results.

(d) Filter bottom HITS authority results.

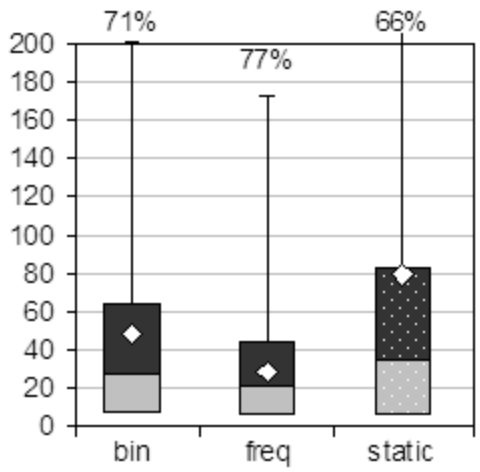
(e) Filter top HITS hub results.

(f) Filter bottom HITS hub results.

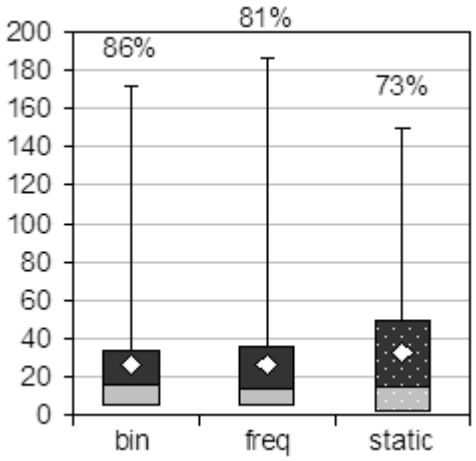
Figure 4.9: The effectiveness measure for the feature location techniques applied to 45 features of Eclipse. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



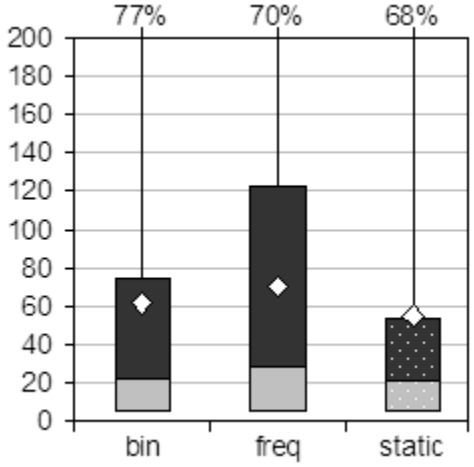
(a) Filter top PageRank results.



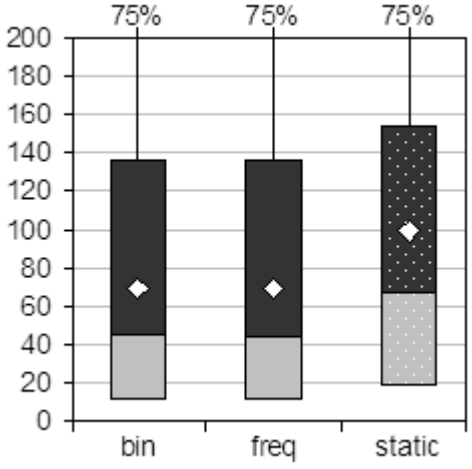
(b) Filter bottom PageRank results.



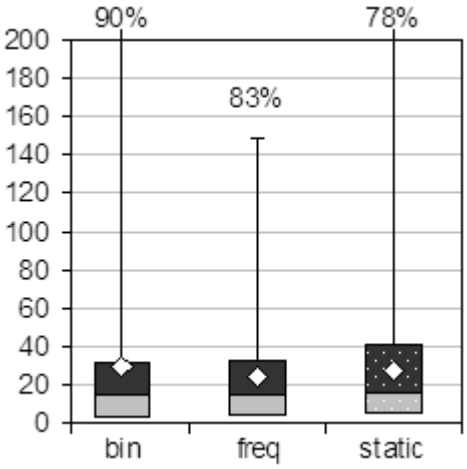
(c) Filter top HITS authority results.



(d) Filter bottom HITS authority results.



(e) Filter top HITS hub results.



(f) Filter bottom HITS hub results.

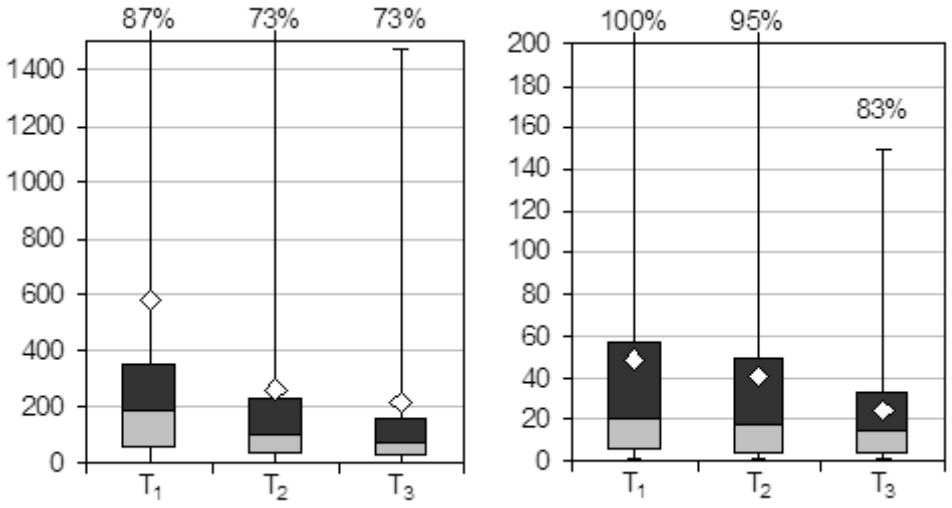
Figure 4.10: The effectiveness measure for the feature location techniques applied to 241 features of Rhino. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

to the dynamic results in Rhino but not Eclipse. Future work will include investigating the circumstances under which a static call graph might yield comparable results.

4.3.5 Discussion

The findings of the evaluation show that combining web mining with an existing feature location technique results is a more effective approach (**RQ1**). Additionally in the context of feature location, HITS is a more effective web mining algorithm than PageRank (**RQ2**). The most effective techniques evaluated were $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$. The results indicate that filtering bottom-ranked hub methods from $IR_{LSI}Dyn_{bin}$'s results is the most effective approach from both the perspective of the position of the first relevant method and of all relevant methods. For instance, for one feature in Eclipse, IR_{LSI} ranked the first relevant method at position 1,696, and for $IR_{LSI}Dyn_{bin}$, the best rank of a relevant method was at position 61. $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$, on the other hand, ranked the first relevant method to the feature at position 24. Filtering the bottom HITS hub methods eliminated 37 false positives from the results obtained by the state of the art technique. Examining the results in detail reveals why. Methods with high hub values call many other methods, while methods that do not make many calls have low hub values. These bottom-ranked hub methods are generally getter and setter methods or other methods that do not make any calls and perform very specific tasks. The $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ technique prunes these methods from the results since they are not relevant to the feature, thus improving effectiveness.

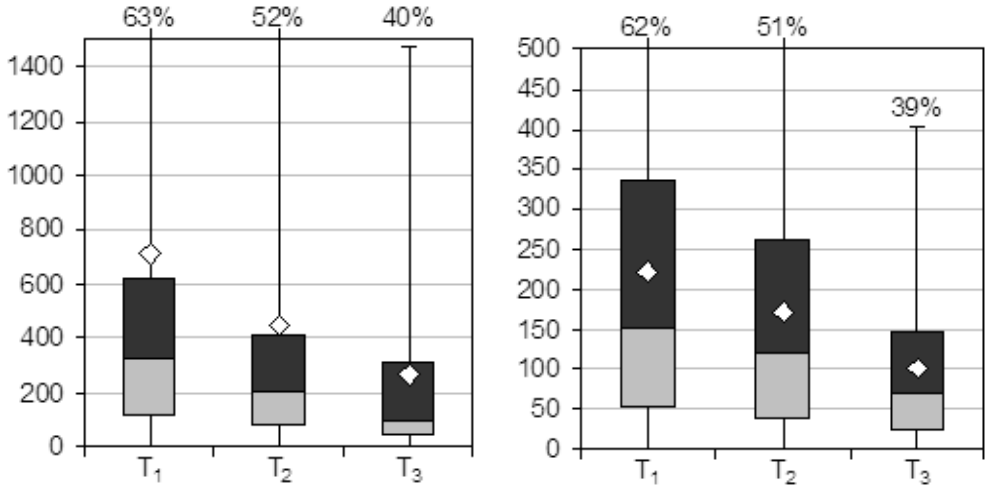
The two most effective techniques remove bottom-ranked hub methods, and these methods tend to be getters and setters. We also compared $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ to a technique that filters out all getter and setter methods from $IR_{LSI}Dyn_{bin}$'s results to see if this simpler filtering heuristic is more effective that using web mining. Figure 4.11 shows the average effectiveness measure of the baseline (T_1), the baseline with getter and setter methods pruned from the results (T_2), and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ (T_3). In both Eclipse and Rhino, removing getters and setters from $IR_{LSI}Dyn_{bin}$'s results is not as effective as



(a) Eclipse. (b) Rhino.

$\mathbf{T_1}$: $IR_{LSI}Dyn_{bin}$
$\mathbf{T_2}$: $IR_{LSI}Dyn_{bin}$ with getters and setters filtered
$\mathbf{T_3}$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$

Figure 4.11: Comparing the effectiveness measure for the baseline technique (T_1), filtering getters and setters from the baseline (T_2), and one of the most effective techniques based on using web mining as a filter (T_3). The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



(a) Eclipse. (b) Rhino.

$\mathbf{T_1}$: $IR_{LSI}Dyn_{bin}$
$\mathbf{T_2}$: $IR_{LSI}Dyn_{bin}$ with getters and setters filtered
$\mathbf{T_3}$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$

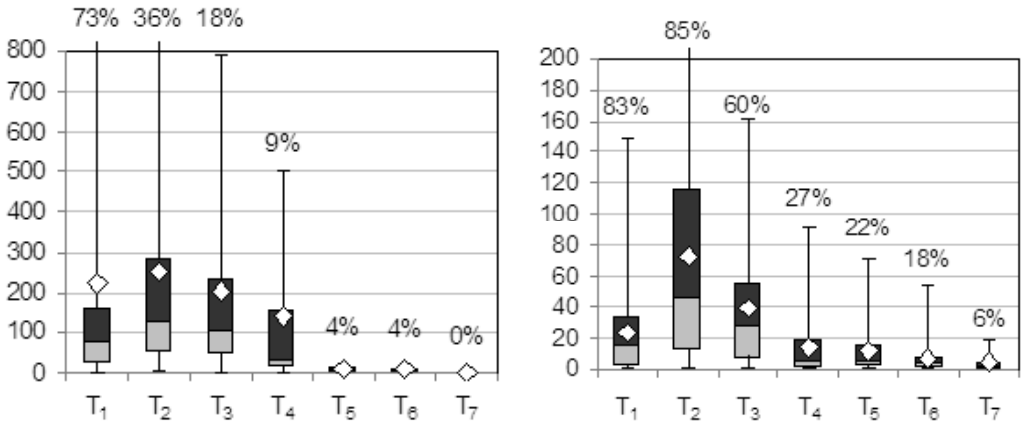
Figure 4.12: Comparing the average position of all gold set methods for the baseline (T_1), filtering getters and setters from the baseline (T_2), and one of the most effective techniques based on using web mining as a filter (T_3). The values above the boxes represent the percent of all the gold set methods the technique could locate.

$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$. Similarly, when considering the ranks of all of a feature's relevant methods, $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ is still the more effective technique, as seen in Figure 4.12. Therefore, using the HITS web mining algorithm and filtering bottom-ranked hub methods eliminates more false positives than simply pruning getter and setter methods.

In addition to investigating the filtering heuristic of eliminating getter and setter methods, we also explored another simplified heuristic in which methods with certain fan-in values are pruned from $IR_{LSI}Dyn_{bin}$'s results. The fan-in of a module is defined as the number of locations from which control is passed in to the module [101] and is derived from a static call graph. Fan-in is similar to web mining. Both count the number of incoming links/calls to a page/method. The difference is that the web mining algorithms are more powerful because they incorporate indirect information. Not only are the number of incoming links counted, the importance of those incoming links are considered. For instance, the PageRank of a page is based upon how many other pages link to the page and the PageRank of those pages. Similarly with HITS, page's authority score is based on how many hubs point to it, not just the total number of pages that link to it. The web mining algorithms are defined recursively (see Sections 4.1.3.2 and 4.1.3.1) to capture this indirect information. Another difference between our work and research on using fan-in is we apply web mining to a dynamic call graph, while fan-in is computed from a static call graph.

Figure 4.13 compares the effectiveness of $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$ to several techniques based on filtering methods with certain fan-in values from $IR_{LSI}Dyn_{bin}$'s results. For instance, T_3 prunes all methods with a fan-in value less than or equal to 2. In both Eclipse and Rhino, the approaches that filter more methods have lower average effectiveness. However, these techniques are only able to locate at least one gold set method for a smaller percentage of all the features. The results are similar when the rankings of all of a feature's methods are considered, as seen in Figure 4.14. Therefore, using fan-in as a filtering heuristic is too naïve and simplistic because it eliminates too many of a feature's relevant methods, unlike using web mining.

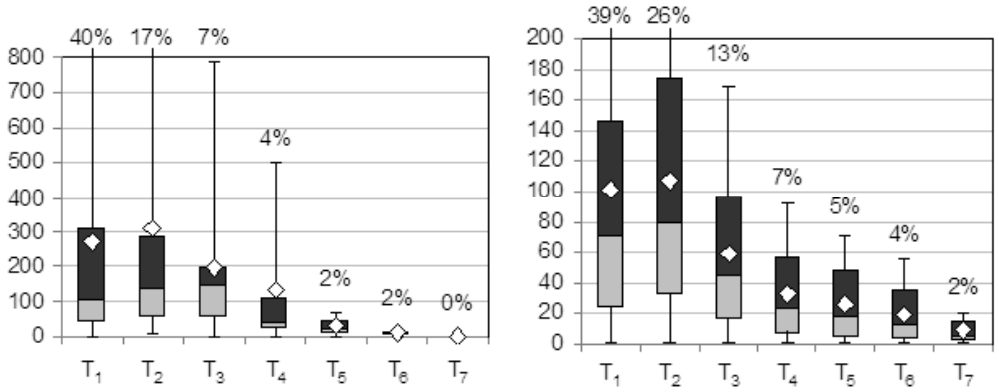
Concerning the use of execution frequency or binary weights, the results do not show a significant difference between the two, nor is one consistently more effective than the



(a) Eclipse. (b) Rhino.

T1 : $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$
T2 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 1
T3 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 2
T4 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 3
T5 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 4
T6 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 5
T7 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 10

Figure 4.13: Comparing the effectiveness measure for one of the most effective techniques based on using web mining as a filter (T_1) and techniques based on filtering methods with certain fan-in values from $IR_{LSI}Dyn_{bin}$'s results (T_2 – T_7). The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



(a) Eclipse. (b) Rhino.

T1 : $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bottom}$
T2 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 1
T3 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 2
T4 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 3
T5 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 4
T6 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 5
T7 : $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 10

Figure 4.14: Comparing the average position of all gold set methods for one of the most effective techniques based on using web mining as a filter (T_1) and techniques based on filtering methods with certain fan-in values from $IR_{LSI}Dyn_{bin}$'s results (T_2 – T_7). The values above the boxes represent the percent of all the gold set methods the technique could locate.

other. However, one observation is that in Rhino, binary weights were more effective, likely because the Rhino traces had many loops which artificially inflated the execution frequencies of many of the methods. Using binary weights avoided this situation.

Each of the analyses used in the data fusion model have their own costs and overheads that must be weighed against the benefits of using the techniques. The main cost associated with LSI is indexing the corpus, which for large corpora can take several minutes, depending on many factors such as the size of the corpus and CPU speed. However, this is a one-time cost and can be performed incrementally when the source code changes [108]. Gathering execution information by collecting traces is probably the most expensive analysis used in the model in terms of both time and space. Tracing a program's execution can impose considerable overhead and significantly slow down execution speed [52]. Collecting a trace of a large system such as Eclipse could take an hour. Additionally, the collected trace will be large in size, possible over a gigabyte (See Table 4.2). Collecting multiple traces requires sufficient storage space to save them all. The final type of analysis used in the framework is web mining. Running the web mining algorithms can take several minutes for a large system. Like indexing with LSI, this is a one-time cost.

4.3.6 Threats to Validity

There are several threats to validity of the evaluation presented in this chapter. Conclusion validity refers to the relationship between the treatment and the outcome and if it is statistically significant. Since no assumptions were made about the distribution of the effectiveness measures, a non-parametric statistical test was used. The results of the test showed that the improvement in effectiveness of most of the web mining based feature location techniques over the state of the art is significant.

Internal validity refers to if the relationship between the treatment and the outcome is casual and not due to chance. The effectiveness measure is based on the position of a feature's first relevant method, and the relevant methods are defined by a gold set. In Eclipse, the gold set was defined by bug report patches. These patches may contain only a subset of the methods that implement a feature, and sometimes the methods were not implemented until a later version. In Rhino, the gold set methods were defined manually by

other researchers who were not system experts. Thus, relevant methods could be missing from the gold sets of each system. This threat is minimized by the fact that the patches were approved by the module owners and the Rhino data has been previously used by other researchers [62, 63].

Another threat to internal validity pertains to the collection of data from IR and dynamic analysis. Information retrieval requires a query. The queries in this evaluation were taken directly from bug reports and documentation. It is possible that the queries used do not accurately reflect the features being located or that the use of different queries with vocabularies more inline with the source code would yield better results. However, using these default queries instead of formulating our own eliminated the introduction of bias. Similarly, execution traces were collected for each feature based on either the bug reports or test cases. The collection of these traces may not have invoked all of a feature's relevant methods or may have inadvertently invoked another feature. This is a threat to validity common to all approaches that use dynamic analysis. The use of test cases distributed with the software reduces this threat since the tests were created by the system's authors.

External validity concerns whether or not the results of this evaluation can be generalized beyond the scope of this work. Two open source systems written in Java were evaluated. Eclipse is large enough to be comparable to an industrial software system, but Rhino is only medium-sized. Additional evaluations on other systems written in other languages are needed to know if the results of this study hold in general.

4.4 Related Work

As discussed in Chapter 2, existing feature location techniques can be broadly classified by the types of analysis they employ, be it static, dynamic, textual, or a combination of two or more of these. This section reviews some of the related work that is most relevant to the work presented in this chapter. We also explain the key differences between our work and the related work.

There are several static approaches to feature location. Chen and Rajlich [39] proposed the use of Abstract System Dependence Graphs (ASDG) as a means of static feature

location, whereby users follow system dependencies to find relevant code. Robillard [176] introduced a more automated static approach that analyzed the topology of a system's dependencies. Harman et al. [98] used hypothesis-based concept assignment (HB-CA) [89] and program slicing to create executable concept slices and found these slices can be used to decompose a system into smaller executable units corresponding to concepts (features) [13]. In our work, instead of using static information, we focus on using textual and dynamic information to get results that are more tailored to a specific feature.

Software reconnaissance [229] is a well-known dynamic approach to feature location. Two execution traces are collected: one that invokes the feature of interest and another that does not. The traces are compared, and methods invoked only in the feature-specific trace are deemed relevant. SPR [5] is another dynamic feature location technique in which statistical hypothesis testing is used to rank executed methods. We also employ dynamic information for feature location, but we use it as a filter to textual information instead of directly identifying a feature's implementation from pure dynamic analysis.

Textual feature location was introduced by Marcus et al. [142] when they applied LSI to source code. The approach has been extended to include relevance feedback [85], where users indicate which results are relevant and a new query is automatically formulated from the feedback. Textual analysis of source code is not limited to LSI. Grant et al. [90] employ Independent Component Analysis (ICA) for feature location. ICA is an analysis technique that separates a set of input signals into statistically independent components. For each method, the analysis determines its relevance to each of the signals, which represent features. Textual feature location is at the foundation of our work. We rely on LSI as opposed to other analyses because LSI is the *de facto* standard.

In addition to these techniques based on a single type of analysis, there are many hybrid approaches. Both SITIR [130] and PROMESIR [160] combine textual and dynamic analysis. Eisenbarth et al. [76] applied formal concept analysis to execution traces and combined the results with an approach similar to ASDGs. This approach involves human input and does not produce ranked results, so we did not include it in our evaluation. Dora [102] and SNIAFL [244] incorporate information from textual and static analysis. Cerberus [62] is the only hybrid approach that combines static, dynamic, and textual analyses. Dora

and Cerberus do not produce ranked results, but SNIAFL does, so future work involves comparing our new techniques to it.

No existing feature location techniques rely on web mining. However, web mining has been used for other program comprehension tasks. Zaidman and Demeyer [240, 239] used the HITS algorithm on a dependence graph of a system weighted with dynamic coupling measures to identify the classes that are most important for understanding the software. Saul et al. [197] also use HITS to recommend related API calls. SPARS-J [106] is a system that analyzes the usage relations of components in a software repository using a ranking algorithm that is similar to PageRank. Components that are generic and frequently reused are ranked highly. Li [127] also uses a variant of PageRank called Vertex Rank Model (VRM) to refine concept bindings found using HB-CA. The VRM works on a dependence graph of concept bindings to identify statements that can be removed from the concept bindings without losing domain knowledge.

Aspect mining is closely related to feature location. The goal of aspect mining is to identify concerns[8] that are scattered throughout a system's modules so that they can be refactored in to their own modules known as aspects. The concerns are not known a priori, whereas in feature location, the features of interest are known before searching begins. Marin et al. use fan-in to identify concerns that can be refactored in to aspects [144, 145]. Methods with high fan-in are called from many different locations within the system, and thus possibly represent a scattered concern. Other aspect mining approaches have employed the idea of data fusion by combining multiple techniques [202] including fan-in, clone detection [32, 33, 203], and natural language analysis [205].

4.5 Conclusion

This work has introduced a data fusion model for feature location. The basis of the model is that combining information from multiple sources is more effective than using the information individually. Feature location techniques based on web mining and approaches using web mining as a filter to information retrieval were instantiated within the model.

[8] A concern is an area of interest or focus in a system. Features can be concerns, but not all concerns are features.

A large number of features from two open source Java systems were studied in order to discover if feature location based on combining IR and web mining is more effective than the current state of the art and which of two web mining algorithms is better suited to feature location.

The results of an extensive evaluation reveal that new feature location techniques based on using web mining as a filter are more effective than the state of the art, and that their improvement in effectiveness is statistically significant. Future work includes instantiating the model with different IR techniques and investigating when static call graphs are acceptable to use. All of the data used to generate the results presented in this chapter is made freely available to other researchers who wish to replicate the case studies.

Chapter 5

Feature Coupling

Chapters 3 and 4 focus on feature location, an important step in the software maintenance process. Programmers use feature location to find the source code that implements features related to a maintenance task. This chapter delves into another step of the maintenance process: revalidating a software system once changes to a feature have been made. Programmers can use impact analysis to determine the effect of their change to a feature. Coupling is one way to perform impact analysis.

Coupling is an important software relationship that has been used for numerous tasks related to software development and maintenance such as predicting software quality [9, 30, 28, 97, 152, 211] and impact analysis [29, 164, 231]. Coupling is primarily measured at the class-level by determining the degree to which two classes in an object-oriented system depend on one another.

Features, also known as concepts or concerns, are functionalities described in a requirements or specification document that have been actualized in a software system [5]. Often, features have implementations that span multiple methods or classes and cannot be modularized due to design decisions [111, 213]. Features are important software entities that transcend the boundaries of classes. Currently, there are no metrics that explicitly capture the coupling between features, and the usefulness of such measures is not known.

In this chapter, we argue that feature-level coupling metrics are needed and show that they are useful. Feature coupling can be used as a predictor of fault-proneness. Just as class coupling has been used in testing [109], if it is known that two features are tightly coupled,

more testing effort can be applied to them to help eliminate bugs. Another example is software maintenance. Many software change tasks are framed in terms of a system's functionalities or features. Since a feature's implementation may be scattered throughout the source code of a software system, programmers may have difficulty determining which other features interact with it. Therefore, changes made to one feature may have unintended consequences for other, seemingly unrelated features, causing improper system behavior. To avoid such situations, feature-level impact analysis should be performed to discover other features that are tightly coupled to the feature undergoing modification. Thus feature coupling metrics are needed to measure the dependencies among features to support a variety of software development and maintenance tasks.

We introduce new feature coupling metrics because current coupling metrics are designed for classes, and features exist at a higher level of abstraction than classes. Features are defined by a portion of a specification and implemented in source code, meaning features are represented by both structured (e.g., source code dependencies) and unstructured (e.g., identifiers and comments in source code) information. Therefore, it is logical to measure feature coupling using both types of data: structured and unstructured. Structured information refers to source code and other related artifacts such as call graphs and program dependence graphs that are ordered in a particular way (i.e., following programming language grammar rules). Unstructured information, on the other hand, refers to internal source code comments, identifier names, and external documentation that encode domain knowledge and design decisions. While comments and documentation can be structured in the form of sentences and organized into sections, they are more free form, unstructured, and do not follow specific rules.

We define feature coupling metrics based on these different sources of information. Structural Feature Coupling (SFC) captures the relationship between two features based on structured information, while Textual Feature Coupling (TFC) measures the coupling between features based on unstructured, textual information in source code using an information retrieval technique called Latent Semantic Indexing (LSI) [59]. In addition, we conjecture that the structured and unstructured data are complimentary, as has been shown elsewhere [62, 102, 160, 164], so we propose to combine SFC and TFC into a hybrid fea-

ture coupling metric called HFC. Hybrid feature coupling can be used when one source of information cannot be completely relied on but programmers still want to incorporate it. For instance, in systems that are poorly structured, more weight can be given to textual information to compensate. Likewise, in software with little or no comments or poorly named identifiers, more weight can be placed on structural information.

This work makes the following research contributions:

1. **Define feature coupling metrics**.

   We formally define coupling metrics for features using structural and textual information. Our metrics are novel and fill a void in the research area that currently lacks feature coupling metrics based on either type of information. We also theoretically validate our metrics and introduce a new dimension to the unified framework for coupling measurement [25].

2. **Demonstrate the relationship between feature coupling and fault-proneness**.

   To demonstrate both the usefulness and applicability of our new feature coupling metrics, we perform three separate case studies. In the first case study, we empirically investigate the relationship between our feature coupling metrics and fault-proneness. In this study, we establish that there is a statistically significant correlation between feature coupling and defects. Our results build on previously published findings [63] that cross-cutting concerns (features)) may cause defects. In essence, our first case study extends prior results by showing that there is also a relationship between coupled features and bugs.

3. **Evaluate the application of feature coupling to impact analysis**.

   We also demonstrate some implications of feature coupling measurement for feature-level impact analysis. Feature coupling is a good starting point for understanding how a change to one feature is likely to affect others. For example, during impact analysis, all features can be ranked by their strength of coupling to the feature being modified. If programmers know that feature A is more tightly coupled to feature B than to feature C, they can expect that a change to A is likely to impact B more than C and

spend more time ensuring B was not adversely affected by the change to A. Also, analyzing related features using coupling metrics can help avoid introducing defects caused by intricate and potentially hidden dependencies [238] among features. We show that feature coupling can be effectively used for impact analysis under certain configurations.

4. **Explore how feature coupling metrics align with developers' opinions**.
   The final way in which we evaluate our new feature coupling metrics is by investigating if they agree with developers' opinions of whether two features are coupled or not. We find that overall, there is agreement between the developers' ratings and our measures, meaning our feature coupling metrics do capture coupling among features as recognized by software developers.

5. **Create tool support for feature coupling**.
   We have developed an Eclipse plug-in for managing features. The tool has functionality to assign portions of code to features and the ability to compute and analyze feature coupling metrics on demand.

The three case studies provide evidence that feature coupling metrics are useful tools programmers can use while performing feature-level software maintenance tasks. Like class coupling measures, they can be used to predict fault-proneness and for impact analysis. These new metrics give programmers greater flexibility because they allow for analysis at a higher level of abstraction than classes.

5.1 Related Work

There are many existing coupling metrics that employ different types of information such as structural, dynamic, textual, or evolutionary. Most of these metrics determine coupling between classes. Our work is distinct from previous research in that it provides a formal way to capture and analyze the strength of coupling among features using various types of information, namely structural and textual. Furthermore, there are no existing metrics that combine information from two or more distinct sources (e.g., structural and textual)

Table 5.1: State of the art in coupling measurement across two dimensions: level of coupling and type of information used to capture the strength of coupling. The metrics proposed in this work are highlighted in boldface.

Coupling Dimension	Structural	Dynamic	Textual	Hybrid	Evolutionary
Class	CBO, RFC, MPC, DAC, C_e, C_a, Information flow coupling, class-attribute interaction, class-method interaction	Dynamic import and export coupling	$CoCC$	Future work	Interaction coupling, Evolutionary coupling, Logical coupling
Feature	***SFC*** ***SFC′***	$DIST$	***TFC*** ***TFC***$_{max}$	***HFC***	Future work

to capture coupling. Table 5.1 summarizes the state of the art in coupling measurement, and we offer a brief overview below.

5.1.1 Structural Coupling Measures

Most existing coupling metrics capture coupling between classes structurally. Coupling Between Objects (CBO) and Response for a Class (RFC) were introduced in Chidamber and Kemerer's suite of object-oriented metrics [42]. According to CBO, two classes are coupled if methods in one class use methods or fields in the other. RFC and RFC_α are counts of a class' methods plus methods that are directly or indirectly [43] invoked by those methods. Li and Henry [126] introduced several class coupling metrics that also utilize structural information. Message Passing Coupling (MPC) between classes A and B is based on the number of static invocations of methods from class B in class A. Data Abstraction Coupling (DAC) is a count of the number of fields in class A that are of type B, while DAC' is a binary version of this metric. There are a wealth of other structural metrics based on class dependencies such as Efferent Coupling (C_e) and Afferent Coupling (C_a) [146].

Briand et al. [26] developed several metrics for measuring the coupling between classes based on structural information from method invocations and the types of fields and parameters. These metrics, plus those by [104] and [64], were reviewed in [25] to build a

unified framework for coupling measurement in object-oriented systems.

Information flow-based coupling (ICP) [124] is a structural measure that takes polymorphism into account. ICP counts the number of methods from a class B invoked in a class A, weighted by the number of parameters. Two alternative versions, $IH-ICP$ and $NIH-ICP$, count invocations of inherited methods and classes not related through inheritance, respectively. Like ICP, some of the coupling measures defined in [29] take polymorphism into account. All of these existing coupling metrics are defined for classes, and therefore are at a lower level of abstraction than our feature coupling metrics.

5.1.2 Other Static Coupling Measures

Other static coupling measures exist along textual and evolutionary dimensions. Poshyvanyk and Marcus [161] define a coupling metric for classes based on textual information extracted from source code identifiers and comments. Their conceptual coupling metric, $CoCC$ (which stands for **Co**nceptual **C**oupling of **C**lasses), captures a new dimension of coupling not addressed by structural or dynamic measures. $CoCC$ is defined for classes, while the metrics we propose are for features. Interaction [248], logical [82], and evolutionary [247] coupling metrics utilize information from repositories to mine information from software artifacts that are frequently co-changed. Such evolutionary information has been used for impact analysis [206], much like coupling metrics. Additionally, coupling metrics have been defined for other applications such as knowledge-based [122] and aspect-oriented [242] systems.

5.1.3 Dynamic Coupling Measures

Arisholm et al. [7] introduced dynamic import and export metrics to capture the coupling between classes at runtime. Dynamic analysis is often used to locate the code associated with features [62, 76, 130, 160, 193, 229] since a feature's behavior can be observed during execution. Currently the only existing feature-level coupling-like metric that we are aware of is based on dynamic information. Wong and Gokhale [233] defined the distance ($DIST$) between two features using an execution slice-based technique. Similar feature metrics have been proposed to dynamically measure certain relationships or dependencies

between features [92, 128] other than coupling. Greevy et al. [93] also created metrics for dynamically measuring the evolution of a feature. Similarly, Giroux and Robillard [88] defined a measure for feature coupling across versions of a system using regression tests since tests typically align with features. The association graph matching similarity measure (AGM) introduced by Kothari et al. [120] is a measure of pair-wise similarity between features based on dynamic call graphs. It has been used to find canonical feature sets [120], feature version similarity [119], and feature implementation overlap [121]. All of these feature metrics solely utilize dynamic information. However, dynamic information may not be sufficient to precisely capture coupling among features. The best way to collect dynamic information is to execute scenarios that exercise only one feature at a time, but developing such scenarios can be difficult, if possible at all [233]. Our metrics are the first to capture feature coupling using structural and textual information, thus avoiding the overhead of collecting execution traces.

5.1.4 Applications of Coupling Metrics

There have been numerous studies showing that coupling is a good predictor of external quality attributes such as fault-proneness [9, 26, 37, 246], maintainability [126], reengineering effort [151], and change-proneness [29]. Other studies have shown that coupling can be used for different tasks [54] such as impact analysis [164, 231], program comprehension [240], reengineering [1], quality assessment [8], reuse [41], change propagation [87], and clone detection [86]. These studies focus on coupling at the class level, while our work examines feature coupling and investigates if it is also useful for predicting fault-proneness and performing impact analysis.

5.2 Analyzing Structured and Unstructured Information in Source Code

The source code of a software system contains structured and unstructured data. The structured data is used primarily by parsers, while the unstructured information (i.e., comments and identifiers) is meant mostly for human readers. The SFC metric that we propose

measures the coupling between two features structurally, drawing on information used by existing class-level coupling metrics. The other feature coupling metric we introduce, TFC, measures the conceptual or textual similarity between two features. Our approach is based on the premise that the unstructured information embedded in source code reflects, to a reasonable degree, the software's domain concepts since existing feature location techniques [130, 142, 160] leverage such textual information to find code that implements features. In order to extract and analyze the unstructured information from source code, we use Latent Semantic Indexing, an advanced information retrieval method. In the remainder of this section, we provide details on how we obtain structured and unstructured information from software.

5.2.1 Structured Information

Class relationships, method invocations, and field references have all been used to compute class coupling [25]. In our work, we focus on methods as the main unit of structural information for several reasons. Working with method-level granularity is common with feature location. Most feature location techniques attempt to find methods associated with features [62, 76, 130, 160, 229] because methods implement functionality in code. Also, several existing class coupling metrics, such as CBO and RFC, use methods only [7, 42, 161], ignoring fields.

Most software engineers are familiar with structural source code information that can be represented in various forms such as a call graph. We use a call graph to add additional information to our structural feature coupling metric. We obtain a method-level call graph using JRipples [35, 156]. We provide more details on how we use this information to capture structural feature coupling in Section 5.3.2.

5.2.2 Unstructured Information

Latent Semantic Indexing identifies relationships between terms and concepts in unstructured text and has been successfully applied to a number of software engineering tasks such as feature location [45, 160, 165, 173], traceability link recovery [3, 57, 100, 108, 137, 148, 220], software measurement [139, 164], and detecting code clones [136, 212]. In LSI,

Table 5.2: Mapping LSI concepts to source code.

LSI Model	Source Code Entities
word	Identifiers and comments extracted from source code comprise a vocabulary set. This set is refined to exclude programming language keywords, stop words, and punctuation. Finally, all compound identifiers are split based on the observed naming conventions. $V = \{w_1, w_2, \ldots, w_v\}$.
document	A method is treated as a document, which can be expressed as n identifiers and comments from a vocabulary and appear in the implementation of a method $m_i = (w_1, w_2, \ldots, w_n)$.
feature	A feature corresponds to a collection of documents representing methods that belong to different classes $F_i = (m_1, m_2, \ldots, m_l)$.
corpus	The software system S consists of a set of classes and features comprised of methods*, $S = (C_1, \ldots, C_z, F_1, \ldots, F_n)$ which forms a corpus $D = (d_1, d_2, \ldots, d_m)$.

*Some of the system's features may be associated with methods and some not. Therefore, the corpus contains features and classes.

a word is a basic unit of discrete data defined to be an item from a vocabulary $V = \{w_1, w_2, \ldots, w_v\}$. A document is a sequence of n words denoted by $d = (w_1, w_2, \ldots, w_n)$, where w_n is the n^{th} word in the sequence. A corpus is a collection of m documents, $D = (d_1, d_2, \ldots, d_m)$. Table 5.2 shows how these LSI concepts are mapped to source code.

The process of applying LSI to source code has three steps. First, the source code must be preprocessed to build a corpus. Second, the corpus is indexed. Third and finally, textual similarities between all pairs of documents (methods) are computed. If two methods use similar terminology and have a high textual similarity, they may implement related concepts and therefore be coupled. Each of these steps is explained in more detail in the following subsections.

5.2.2.1 Build the Corpus

A corpus represents all the words found in each document of a body of text. A document can be a sentence, a paragraph, a chapter, or in the case of source code, a method, a class, or a package. To build a corpus for the source code of a software system, a document granularity must first be chosen. In our work, we use methods as documents. Next, the text of each document must be preprocessed before being included in the corpus. There are

several options for preprocessing, such as removing stop words and programming language keywords, splitting compound identifiers, including or excluding comments, and performing or not performing stemming. Stemming [158] reduces words to their root form, such that "stemming" and "stemmed" would become "stem." For every corpus created in this work, stop words (e.g., *the*, *of*) and programming language keywords (e.g., *public*, *for*, *try*) were removed and compound identifiers were split.

5.2.2.2 Index the Corpus

The central concept of LSI is that the information about the contexts in which a word appears or does not appear provides a set of mutual constraints that determines the similarity of meaning of sets of words (documents) to each other. LSI indexes a corpus and generates a real-valued vector description for each document based on the vector space model (VSM) [196]. LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with VSM approaches. Some words appear in the same contexts, and an important part of word usage patterns is blurred by accidental and inessential information. The method used by LSI to capture essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix (words by documents) decomposed using singular value decomposition (SVD) [195]. The word $\times$ document matrix holds term frequency-inverse document frequency (tf-idf) values which assess how important a particular word is to a given document. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of a matrix without serious loss of specificity. Typically, the word by document matrix is very large and quite sparse. SVD is applied to the word-by-document matrix to eliminate noise.

5.2.2.3 Compute Textual Similarities

Once the corpus is indexed, the similarities between documents can be computed by taking the cosine between their corresponding vectors. The textual similarity between two documents (methods) m_i and m_j is defined as the cosine between vectors vm_i and vm_j, corresponding to m_i and m_j after dimensionality reduction is applied. Just as cosine values

range from -1 to 1, so do textual similarities. The closer a value is to one, the more similar the texts of the documents/methods are. Note that textual similarities are symmetric; the similarity between m_i and m_j is the same as the similarity between m_j and m_i. In Section 5.3.8, an example of how to compute textual feature coupling using the textual similarities between two features is given.

5.3 Using Structural and Textual Information for Feature Coupling

Our approach to measuring feature coupling is based on two main ideas: 1) features are entities that are coupled at a higher level of abstraction than methods and classes and 2) coupling can be measured in multiple ways by using structured and unstructured (textual) information. Features are domain concepts implemented in a system, and their implementations are often scattered across a system's classes [63]. Therefore, features exist at a level of abstraction outside of or above classes in object-oriented languages. As described in Section 5.1, there exists an abundance of class coupling metrics that rely on structural dependencies and some that utilize textual information to measure class coupling. These metrics are useful and important because they capture essential forms of coupling. However, since features transcend class boundaries, we propose and define metrics that comprehensively capture and measure feature coupling using both structural and textual information.

5.3.1 System Representation

To define structural and textual feature coupling metrics, we first define a representation of a software system.

Definition 1: (System, Classes, Methods)

A system S is an object-oriented software system. S has a set of classes $C = \{c_1, c_2, \ldots, c_n\}$. The number of classes in S is $n = |C|$. A class has a set of methods. For each class $c \in C$, let $M_c = \{m_1, m_2, \ldots, m_z\}$ be the set of methods implemented in c, where $z = |M_c|$ is the number of methods in c. The set of all methods in the system S is defined as M_S.

Definition 2: (Feature)

A feature f is a requirement, functionality, or behavior described in the specification of a system S. We base this definition on Eaddy et al.'s [63] well-established model for representing cross-cutting concerns (features)[1]. A system S has a set of features $F = \{f_1, f_2, \ldots, f_p\}$ where $p = |F|$. A feature f is implemented by a set of methods $M_f \subseteq M_S$. The methods of M_f may belong to multiple classes. A method may belong to several features, and a feature may have methods that belong to other features as well.

5.3.2 Structural Feature Coupling

We define structural feature coupling metrics using our representation of a system, features, and methods.

Definition 3: (Structural Feature Coupling – SFC)

The structural feature coupling (SFC) between features f_a and f_b, implemented by the methods in sets M_a and M_b, respectively, is defined as the ratio of the number of methods shared by the features to the total number of methods associated with the two features.

$$SFC(f_a, f_b) = \frac{M_a \cap M_b}{M_a \cup M_b} \tag{5.1}$$

We only consider features with non-empty methods sets to avoid a potential division by zero. SFC uses structured information to capture feature coupling by measuring the degree to which two features share code.

Definition 4: (Structural Features Coupling Prime – SFC′)

Instead of solely basing coupling on the methods that implement two features, an alternative is to consider the first order structural dependencies of those methods to also be associated with the features. Dependencies are taken into account in some existing coupling metrics (e.g., RFC), plus they are often traversed for maintenance, feature location, and program comprehension tasks. Therefore, we include the static callers and callees of a feature's methods in a variant SFC, which we coin SFC'.

Let f_a and f_b be features implemented by the methods in sets M_a and M_b respectively. Let $M'_a \supseteq M_a$ and $M'_b \supseteq M_b$ be the set of methods that implement features f_a and f_b,

[1] Eaddy et al.'s [63] definition is more general than ours, encompassing fields and concern hierarchies.

respectively, plus the methods that are first order structural dependencies of the methods in M_a and M_b. That is, M'_a and M'_b include the methods that call or are called by the methods in M_a and M_b. The structural feature coupling prime (SFC') is defined as the number of methods shared by two features over the total number of methods associated with both features.

$$SFC'(f_a, f_b) = \frac{M'_a \cap M'_b}{M'_a \cup M'_b} \tag{5.2}$$

Thus, SFC' incorporates additional structured information in the form of dependencies to measure feature coupling. Both SFC and SFC' are normalized, i.e., they have values in the range $[0, 1]$. The closer the value is to one, the stronger the structural coupling between the features.

5.3.3 Textual Feature Coupling

We define textual feature coupling metrics based on unstructured, textual information found in source code. In order to define a metric for the textual coupling between features, we first define the conceptual similarity between two methods as well as between a method and a feature. These measures are building blocks needed to define our textual feature coupling metric.

Definition 5: (Conceptual Similarity between Methods – CSM)

As defined in [138], the conceptual similarity, also known as the textual similarity, between methods $m_i \in M_S$ and $m_j \in M_S$ is $CSM(m_i, m_j)$ where

$$CSM(m_i, m_j) = \frac{vm_i^T vm_j}{|vm_i|_2 \times |vm_j|_2} \tag{5.3}$$

$CSM(m_i, m_j)$ is the cosine between vectors vm_i and vm_j, corresponding to m_i and m_j after indexing. As defined, the value of $CSM(m_i, m_j) \in [-1, 1]$. In order to comply with the non-negativity property of coupling [27], if $CSM(m_i, m_j) \leq 0$, we redefine $CSM(m_i, m_j) = 0$. CSM measures the textual similarity of two methods, but most features are composed of more than one method. Next, we define the conceptual similarity between a single method and a feature.

Definition 6: (Conceptual Similarity between a Method and a Feature – CSMF)

Let f_a and f_b be two distinct features in S. Each feature has a set of methods $M_a =$

$\{m_{a1}, m_{a2}, \ldots, m_{ax}\}$, where $x = |M_a|$ and $M_b = \{m_{b1}, m_{b2}, \ldots, m_{by}\}$, where $y = |M_b|$. Between every pair of methods, there is a similarity measure $CSM(m_a, m_b)$. The textual similarity between a method m_a from f_a and a feature f_b is:

$$CSMF(m_a, f_b) = \frac{\Sigma_{q=1}^{y} CSM(m_a, m_{bq})}{y} \tag{5.4}$$

which is the average of the textual similarities between a method m_a and all methods in feature f_b. Now that we have a measure of the textual similarity of one method to a feature, we can define the textual similarity among all the methods of two features, i.e. their textual coupling.

Definition 7: (Textual Feature Coupling – TFC)

Let f_a and f_b be two distinct features in S. The textual coupling between f_a and f_b is:

$$TFC(f_a, f_b) = \frac{\Sigma_{l=1}^{x} CSMF(m_{al}, f_b)}{x} \tag{5.5}$$

which is the average of the textual similarity measures between all unordered pairs of methods from feature f_a and f_b. $TFC(f_a, f_b)$ is a measure of the textual coupling between the two features. This definition guarantees that the coupling between two features is symmetric.

Definition 8: (Maximum Textual Feature Coupling – TFC_{max})

In [138], a variant of the conceptual class coupling metric was used in which only the highest textual similarities between methods of a class are considered. Similarly, we define such an alternative measure for textual feature coupling. We refine TFC to only capture the strongest textual similarity between features. Under this definition, the textual similarity between a method m_a and a feature f_b is computed using the maximum value of $CSM(m_a, m_b)$ such that $CSMF_{max}(m_a, f_b) = max\{CSM(m_a, m_b)\}$ for all $m_b \in M_b$. With this variation, the maximum textual coupling (TFC_{max}) between two features f_a and f_b is:

$$TFC_{max}(f_a, f_b) = \frac{\Sigma_{l=1}^{x} CSMF_{max}(m_{al}, f_b)}{x}. \tag{5.6}$$

5.3.4 Hybrid Feature Coupling

Definition 9: (Hybrid Feature Coupling – HFC)

Structural information aligns with a program's structured information (e.g., source code) while unstructured, textual information aligns with domain concepts (e.g., requirements). We combine structural and textual information into a single feature coupling metric to take advantage of this complementary relationship. To combine structural and textual coupling measured by SFC and TFC, we rely on an affine transformation. Thus, the hybrid coupling between features f_a and f_b is defined as:

$$HFC(f_a, f_b) = w_{SFC} * SFC(f_a, f_b) + w_{TFC} * TFC(f_a, f_b). \tag{5.7}$$

The structural and textual weights, w_{SFC} and w_{TFC}, are values between zero and one and are chosen such that the sum of the weights is equal to one. The higher the weight, the more preference is given to that metric. Affine transformations have been used to combine different types of information for class cohesion [58], feature location [160], and identifying duplicate bug reports [223]. We chose this straightforward means of combining the two metrics because we were interested in investigating, in a controllable fashion, whether combining structural and textual information captures new facets of feature coupling.

5.3.5 Theoretical Evaluation

Our feature coupling metrics comply with the five mathematical measurement properties proposed by Briand et al. [27]: non-negativity, null value, monotonicity, merging of modules, and merging of unconnected modules. Both our structural and textual feature coupling measures assume non-negative values. SFC and SFC' are based on the cardinality of sets and therefore their minimum value is zero. By redefining CSM to always produce a value greater than or equal to zero, TFC and TFC_{max} comply with the non-negativity property. Since HFC is based on an affine transformation of SFC and TFC, it also obeys the property. Additionally, when there is no relationship between two features, our metrics return a measurement of zero, meeting the null value property. To fulfill the monotonicity property, when a new method is added to a feature that is shared by another feature or had a strong

textual similarity to methods in another feature, our coupling metrics increase instead of decreasing. Finally, the coupling obtained after merging two features is not greater than the sum of the coupling of the two original features; thus the final two properties are met.

5.3.6 Classification within the Unified Framework for Coupling Measurement

Briand et al. [25] classified coupling metrics along a number of criteria such as the type of coupling, the direction of coupling, direct vs. indirect coupling, inheritance-based vs. non-inheritance-based coupling, and domain of measurement. Our metrics are new and rely on several mechanisms not currently supported by the unified framework. The framework needs to be expanded to include a new level of granularity for features. Additionally, at the time the framework was created, class coupling was measured using structural information only. Since its definition, conceptual/textual coupling [161] has been established, which necessitates the introduction of a new dimension to the framework that takes into account textual information. We extend the unified framework for coupling measurement to account for feature-level granularity and textual coupling and classify our metrics within the expanded version.

All of the existing coupling measures surveyed for the framework take into account structural information to define the type of connectivity between elements of a class. The existing coupling metrics were classified according to seven different types of connectivity, listed in Table 5.3. We extend the types of connection to include structural and textual relationships between methods of features. We also classify our metrics using the other criteria proposed by Briand et al. [25]. Import coupling refers to a class that uses (imports) another class, while export coupling denotes a class that is used by another. Our feature coupling metrics measure both import and export coupling. Direct and indirect coupling measure direct connections and indirect connections, respectively. SFC, TFC, TFC_{max}, and HFC are all direct measures, but SFC' is indirect because it also includes callers and callees of a feature's methods. Currently, inheritance is not explicitly considered in our feature coupling measures and only methods of a class that are implemented or overloaded in a class are associated with features. Therefore, all of our feature coupling metrics can

Table 5.3: Types of connection, a dimension of the unified framework for coupling measurement.

#	Element 1	Element 2	Description	Measures
1	Attribute a of class c	Class d, $d \neq c$	Class d is of type a	DAC, DAC', class-attribute
2	Method m of class c	Class d, $d \neq c$	Class d is the type of a parameter of m or m's return type	class-method interaction
3	Method m of class c	Class d, $d \neq c$	Class d is the type of a local variable of m	
4	Method m of class c	Class d, $d \neq c$	Class d is the type of a parameter of a method invoked by m	
5	Method m of class c	Attribute a of class d, $d \neq c$	m references a	CBO, CBO', COF
6	Method m of class c	Method m' of a class d, $d \neq c$	m invokes m'	CBO, CBO', RFC, RFC_α, MPC, COF, ICP, $NIH-ICP$, $IH-ICP$, method-method interaction
7	Class c	Class d, $d \neq c$	High level relationships between classes	
8	**Method m of feature f**	**Method m' of feature g, $g \neq f$**	**m is the same as m'**	***SFC*, *SFC'***
9	**Method m of feature f**	**Method m' of feature f**	**m and m' are textually similar**	***TFC*, TFC_{max}**

Table 5.4: Mapping coupling measure to domain.

Domain	Measures
Attribute	
Method	ICP, $NIH-ICP$, $IH-ICP$
Class	CBO, CBO', RFC, RFC_α, MPC, COF, class-attribute interaction, class-method interaction, method-method interaction, CoCC
Set of Classes	ICP, $NIH-ICP$, $IH-ICP$
Feature	***SFC*, *SFC'*, *TFC*, TFC_{max}, *HFC***
System	COF

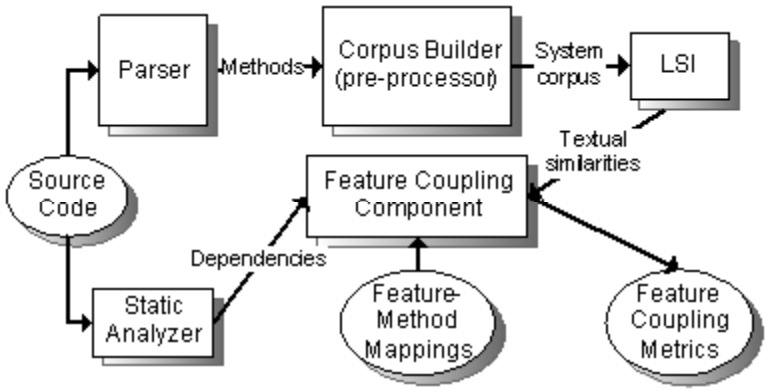


Figure 5.1: Architecture of the feature coupling component of FLAT^3.

be classified as non-inheritance based. Finally, the dimension that most distinguishes our coupling metrics from existing ones is the domain of measurement. Table 5.4 lists the five domains identified by Briand et al. [25] and their associated measures. We extend the unified framework for coupling measurement with a new dimension, the feature domain, and our metrics belong in this classification.

5.3.7 Measurement Tool

We have developed tool support for feature coupling measurement. FLAT^3 (**F**eature **L**ocation and **T**extual **T**racing **T**ool), which is overviewed in Figure 5.1 and described in detail in Chapter 6, is an Eclipse plug-in based on ConcernMapper[2] and ConcernTagger[3] that supports mapping features to source code and the computation of feature coupling metrics. Users can manually associate features with source code or use an embedded feature location technique based on prior research [130]. Alternatively, feature-method mappings can be imported from existing models [143, 182] or tools such as ConcernMapper or ConcernTagger. If the source code or mappings are changed in successive versions of a system, the data given to FLAT^3 must also be updated.

Admittedly, the cost of mapping features to code can be expensive, but research areas such as feature location are focused on automatically recovering such mappings. For instance, Ratiu and Deissenboeck [169, 170] have developed a formal framework for mapping

[2] `http://www.cs.mcgill.ca/~martin/cm/`
[3] `http://www.cs.columbia.edu/~eaddy/concerntagger/`

domain concepts to program elements. Also, some integrated development environments like IBM's Jazz[4] have embedded automatic traceability functionalities for requirements and bug fixes that could be leveraged. These techniques and tools can ease the burden of creating feature-method mappings.

Based on the mappings of features to code, our feature coupling metrics can be computed. First, the source code of a system is parsed into methods. Then, the text of the methods is pre-processed to form the documents of the corpus. Pre-processing always removes stop words and programming language keywords and splits compound identifiers. Options include removing comments from the corpus and performing stemming. Then, LSI is used to create a word-by-document matrix that describes the distribution of terms in the methods of the corpus. Through the use of SVD, a semantic subspace is constructed in which each method from the corpus is represented as a vector. The cosine between two vectors (i.e., CSM) is a measure of the textual similarity between two methods. Given the similarities between methods and the mappings of features to methods, FLAT^3 can compute TFC. To compute SFC, the tool simply requires feature-method maps as well as dependency information.

5.3.8 An Example of Measuring Feature Coupling

We provide an illustrative example of how SFC and TFC are calculated. The example is taken from our evaluation of Rhino, a Java implementation of JavaScript, and two of its features are type conversions ToString (f_{string}) and ToObject (f_{object}). Feature f_{string} is implemented by four methods ($M_{string} = \{m_{s1}, \ldots, m_{s4}\}$), and f_{object} is implemented by eight methods ($M_{object} = \{m_{o1}, \ldots, m_{o8}\}$). Note that m_{s2} is the same as m_{o8}.

The structural coupling between these two features is straightforward to compute. $SFC(f_{string}, f_{object}) = 1/11 = 0.09$ because the two features have one method in common out of 11 total. Our metric captures the weak structural coupling between f_{string} and f_{object}. The two features are concerned with converting an argument, and the only method they share deals with determining the type of the argument before the conversion.

[4] `http://jazz.net/`

Table 5.5: Textual similarities (CSM values) between methods of Rhino's ToString and ToObject features.
m_{s1}: ScriptRuntime.toString(Object);
$m_{s2} - m_{o8}$: FunctionObject.convertArg(...);
m_{s3}: Context.toString(Object);
m_{s4}: NativeRegExpCtor.setInstanceIdValue(...);
$m_{o1} - m_{o5}$: ScriptRuntime.toObject(*);
$m_{o6} - m_{o7}$: Context.toObject(*)

	$\boldsymbol{m_{o1}}$	$\boldsymbol{m_{o2}}$	$\boldsymbol{m_{o3}}$	$\boldsymbol{m_{o4}}$	$\boldsymbol{m_{o5}}$	$\boldsymbol{m_{o6}}$	$\boldsymbol{m_{o7}}$	$\boldsymbol{m_{o8}}$
$\boldsymbol{m_{s1}}$	0.6	0.24	0.54	0.68	0.36	0.23	0.19	0.24
$\boldsymbol{m_{s2}}$	0.28	0.25	0.27	0.33	0.25	0.48	0.37	1.0
$\boldsymbol{m_{s3}}$	0.17	0.16	0.18	0.22	0.18	0.57	0.28	0.42
$\boldsymbol{m_{s4}}$	0.06	0.08	0.06	0.07	0.05	0.13	0.11	0.19

To compute textual coupling, the following formula is used: $TFC(f_{string}, f_{object}) = (CSMF(_{ms1}, f_{object}) + CSMF(m_{s2}, f_{object}) + CSMF(m_{s3}, f_{object}) + CSMF(m_{s4}, f_{object}))/4$. $CSMF(m_{s1}, f_{object})$ is the average of the textual similarities between method m_{s1} and all methods in f_{object} such that $CSMF(m_{s1}, f_{object}) = (CSM(m_{s1}, m_{o1}) + CSM(m_{s1}, m_{o2}) + \ldots + CSM(m_{s1}, m_{o8}))/8$. The textual similarities between methods are shown in Table 5.5. The values in Table 5.5 are the CSM values. Thus $CSMF(m_{s1}, f_{object}) = (0.60 + 0.24 + 0.54 + 0.68 + 0.36 + 0.23 + 0.19 + 0.24)/8 = 0.39$, $CSMF(m_{s2}, f_{object}) = 0.40$, $CSMF(m_{s3}, f_{object}) = 0.27$, and $CSMF(m_{s4}, f_{object}) = 0.09$. Finally, $TFC(f_{string}, f_{object}) = (0.39 + 0.40 + 0.27 + 0.09)/4 = 0.29$. The textual coupling between f_{string} and f_{object} is stronger than the structural coupling. The two features do use some common identifiers such as "Number," "Object," "ScriptRuntime," and "val," but otherwise, they have their own vocabularies.

To calculate the hybrid coupling between these two features, the weight given to each type of coupling needs to be established. If $w_{SFC} = 0.5$ and $w_{TFC} = 0.5$, then the hybrid feature coupling is computed as $HFC(f_{string}, f_{object}) = 0.5 * 0.09 + 0.5 * 0.29 = 0.19$.

5.4 Case Studies

The purpose of our evaluation is to assess the usefulness of our new feature coupling metrics as well as to show that they have a practical application. We perform three assessments of the metrics, each targeting a different aspect of their utility or applicability. In the first case

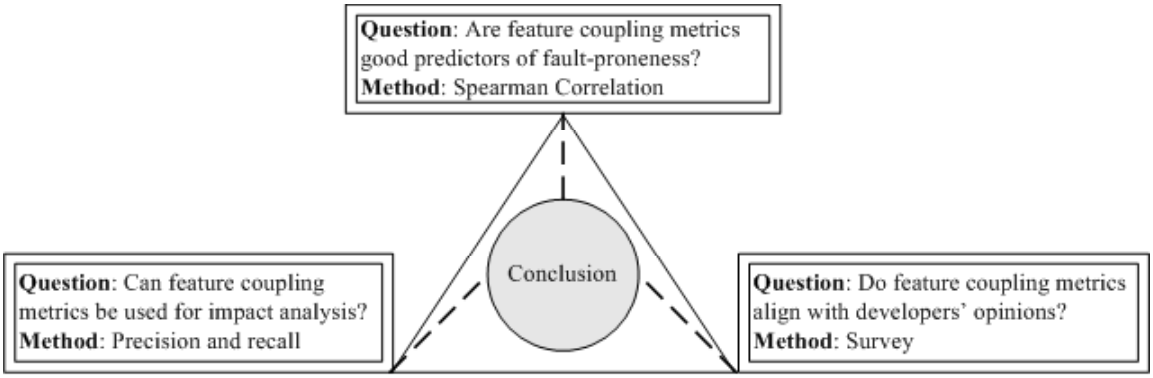


Figure 5.2: Data triangulation evaluation approach.

study, we explore the relationship between feature coupling and fault-proneness. To that end, we calculate the correlation between the metric values and bugs for all unique pairs of features in two software systems. If there is a high correlation between a feature coupling metric and defects, then that metric may serve as a useful predictor of fault-proneness among features. For our second case study, we examine the application of feature coupling metrics for impact analysis. If feature coupling metrics help determine other features likely to be affected by a change to a feature undergoing modification, then these new measures are helpful in the context of impact analysis. Finally, our third case study involves testing if the feature coupling metrics align with developers' opinions about which features are coupled or not. We carry out a survey in which 31 programmers rated the strength of coupling between 16 pairs of features from three different software systems.

By considering the results of three evaluations, we can come to a stronger conclusion about the usefulness of feature coupling metrics than if we had used only one assessment. This idea of synthesizing data from multiple analyses is known as data triangulation [237]. The advantage of such an approach is that by corroborating multiple sources of evidence, any findings or conclusions are likely to be more valid. Figure 5.2 summarizes our data triangulation approach, and in the following sections we provide the details and results of each part of our evaluation.

5.4.1 Subject Systems and Data Sets

To be able to compute our feature coupling metrics, we required mappings of features to the methods that implement them in a given software system. Obtaining this information from a single developer is difficult, time-consuming, and biased [183]. These factors led us to select several existing data sets made available by Eaddy et al. [63] in which multiple researchers compiled mappings of features to code. We used the information in these data sets to compute our feature coupling metrics, and we consider these data sets to be reliable since they have been previously used in other studies [62, 63]. Since we utilized previously published data, our study is reproducible; we invite other researchers to replicate our work. All of our data and results are provided in an online appendix[5].

The first data set we use is dbViz[6] version 0.5, an open-source database visualization tool written in Java. The system is comprised of 12,700 LOC (lines of code), 93 classes, and 554 methods. We also utilize the Rhino data set. Rhino[7] is a Java implementation of JavaScript consisting of approximately 32,000 LOC, 138 classes, and over 1,800 methods. The final data set we use is iBatis[8] version 2.3, an object-relational mapping tool written in Java that has 13,300 LOC, 212 classes, and over 1,800 methods.

The data sets include mappings of program elements to features. Eaddy et al. [63] identified 13 features from dbViz's use cases, 411 features in Rhino from the ECMAScript specification[9] of JavaScript, and 132 features for iBatis. For each feature in the data sets, the code associated with it was manually identified using the prune dependency rule: "A program element is relevant to a [feature] if it should be removed, or otherwise altered, when the [feature] is pruned" [63]. In other words, to assign code (methods and fields) to the features they implement, Eaddy et al. [63] considered a scenario where a feature was to be removed from a system and attempted to remove as much relevant code as possible without affecting other features. While the data sets map some fields to features, we excluded field mappings from our evaluation because our model does not currently support them.

[5] http://www.cs.wm.edu/semeru/data/ese-feature-coupling/
[6] http://jdbv.sourceforge.net/dbViz
[7] http://www.mozilla.org/rhino
[8] http://ibatis.apache.org/
[9] http://www.ecma-international.org/publications/standards/Ecma-262.htm

If the features identified in the data sets are well encapsulated by classes, then measuring feature-level coupling is without merit. To check if the features in the three data sets are implemented in multiple classes, we calculated the average and median number of classes per feature (this data is also available in [61]). In dbViz, features are located in nine classes, on average, with two being the minimum, 21 the maximum, and 9 the median. The average number of classes per feature in Rhino is four, with a minimum of one, a maximum of 67, and a median of 2. Finally, iBatis' features are implemented in six classes on average, with a minimum of one, a maximum of 128, and a median of 3. Since most of the features from the three data sets are implemented in multiple classes, traditional class-level coupling metrics are not able to capture the dependencies between features. Therefore, metrics at a higher level of abstraction, such as feature coupling metrics, are needed.

The data sets also include defect information. We use this data on bugs and where they occur in our first two case studies. In dbViz, 47 bugs are mapped directly to features. Each feature has at least two bugs associated with it, and on average, a feature has 4.7 bugs. In Rhino, 149 bugs are mapped to program elements. Of the 411 features, 344 have bugs, and each feature has 6.4 bugs on average. The publically available data sets did not include defect data for iBatis. If a method was modified to fix a bug, that method is associated with that bug. Transitively, if a feature is associated with a method, and that method was changed to fix a bug, then that bug is mapped to that feature. See [63] for the complete details on how the mappings were obtained.

5.4.2 Case Study Settings

In Section 5.2.2.1, we explained the process of building a corpus in order to obtain textual similarities between methods. There are several options for building a corpus; comments can be included or excluded and text can be stemmed or not. Comments embed additional domain knowledge within the source code of a system. Their inclusion, or exclusion, from a corpus can have an impact on the textual similarities between methods [138]. Stemming reduces words to their root, thus potentially increasing the textual similarity of two documents. Both of these options have implications for textual feature coupling. One of the secondary goals of our evaluation is to discover the optimal configuration for measuring

textual feature coupling. We generated different versions of the corpus of a software system in order to explore the effect of corpus creation on feature coupling. The four corpus versions we created were 1) comments included but without stemming ($c - ns$), 2) comments included and stemming performed ($c - s$), 3) without comments but with stemming ($nc - s$), and 4) comments excluded and no stemming ($nc - ns$). These corpora represent all possible combinations of the preprocessing options for comments and stemming. We consider the $c - ns$ corpus to be the default. For one system in which external documentation was available (Rhino), we made a fifth corpus ($c - ns + d$). This corpus included source code text including comments, the external documentation's text, and words were not stemmed. The documentation is simply added to the corpus as more text; it is not mapped to source code. This augmented corpus was then used by LSI to compute similarities. The idea behind including documentation is that it encodes additional domain knowledge which may bolster the textual information in source code.

5.4.3 The Relationship Between Feature Coupling and Faults

To investigate the relationship between feature coupling and fault-proneness, we performed an empirical study. We conjecture that since features can be implemented in classes and methods dispersed throughout a system, the impact of changes to features can be difficult to determine, possibly leading to faults or system failures. Therefore, we hypothesize that the more coupled two features are, the more likely they are to share a bug. More formally, we seek to evaluate the following hypotheses.

$\boldsymbol{H}_0$ The null hypothesis is that there is no significant correlation between the strength of coupling of two features and the number of bugs they have in common.

$\boldsymbol{H}_1$ The alternative hypothesis is that there is a statistically significant correlation between the strength of coupling of two features and the number of bugs they share.

If H_1 is true, it means that if programmers are aware of other features that are highly coupled to the one of interest, they can potentially prevent the introduction of tedious, feature-related faults. To test our hypotheses, we computed feature coupling metrics between all pairs of features in dbViz and Rhino. Additionally, we counted the number of

Table 5.6: Descriptive statistics of the feature coupling metrics.

System	Metric	Max	75%	Med.	25%	Min	μ	σ
dbViz	***SFC***	0.85	0.08	0.04	0.01	0	0.08	0.15
	SFC'	0.92	0.4	0.32	0.25	0	0.33	0.19
	TFC	0.22	0.09	0.08	0.06	0.02	0.08	0.04
	$\boldsymbol{TFC_{max}}$	0.97	0.46	0.35	0.29	0.08	0.41	0.2
	HFC	0.53	0.09	0.06	0.04	0.01	0.08	0.09
Rhino	***SFC***	1.0	0.0	0.0	0.0	0.0	0.02	0.11
	SFC'	1.0	0.05	0.01	0.0	0.0	0.06	0.16
	TFC	1.0	0.22	0.13	0.09	0.0	0.19	0.17
	$\boldsymbol{TFC_{max}}$	1.0	0.44	0.23	0.14	0.0	0.32	0.24
	HFC	1.0	0.12	0.07	0.04	0.0	0.11	0.12
iBatis	***SFC***	1.0	0.0	0.0	0.0	0.0	0.01	0.05
	SFC'	1.0	0.02	0.0	0.0	0.0	0.03	0.09
	TFC	0.99	0.13	0.09	0.06	0 .0	0.11	0.1
	$\boldsymbol{TFC_{max}}$	1.0	0.33	0.22	0.13	0.0	0.26	0.18
	HFC	0.91	0.07	0.04	0.03	0.0	0.06	0.07

bugs shared by two features for all feature pairs in each system. Then, we computed the Spearman rank order correlation between the metrics and defects.

For each system, we computed five feature coupling metrics for each pair of features: SFC, SFC', TFC, TFC_{max}, and HFC. TFC and TFC_{max} are based on the default corpus, and for HFC, we placed equal weight on structural and textual information. We also refer to this instance of HFC as $S_{0.5}T_{0.5}$, indicating a structural weight of 0.5 and a textual weight of 0.5. For each of these feature coupling metrics, we investigated their relationship with faults. Table 5.6 summarizes the descriptive statistics for the feature coupling measures. We list the maximum (max), minimum (min), inter-quartiles (75%, median, 25%), mean (μ), and standard deviation (σ). The values are based on the 78 unique pairs of features in dbViz, 84,255 pairs in Rhino, and 13,041 pairs in iBatis.

In addition to computing coupling metrics for each pair of features, we also determined the number of defects shared by any two features. We considered a bug to be associated with a pair of features if any methods mapped to the features are also associated with the bug. Consider the example in Figure 5.3. Bug_1 is mapped to methods m_1, m_2, and m_3, while Bug_2 is associated with m_4 and m_5. Feature f_a is implemented by methods m_1 and m_3, while f_b is mapped to m_3, m_4, and m_5. f_a is associated with Bug_1 because its two methods are associated with the defect. Likewise, f_b is associated with Bug_1 and Bug_2.

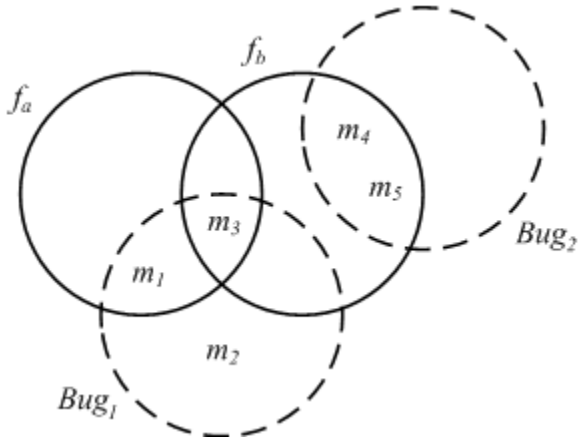


Figure 5.3: An example showing how shared bugs between features f_a and f_b were determined.

Table 5.7: Spearman correlation coefficients for dbViz and Rhino. All values are statistically significant at the one percent level (two-tailed). The sample size (number of feature pairs) is 78 for dbViz and 84,255 for Rhino.
$c - ns$: comments, no stemming
$c - s$: comments stemming
$nc - s$: no comments, stemming
$nc - ns$: no comments, no stemming
$c - ns + d$: comments, no stemming, external documentation

	dbViz				**Rhino**				
Metric	***c-ns***	***c-s***	***nc-s***	***nc-ns***	***c-ns***	***c-s***	***nc-s***	***nc-ns***	***c-ns+d***
SFC	0.38	0.38	0.38	0.38	0.62	0.62	0.62	0.62	0.62
SFC'	0.35	0.35	0.35	0.35	0.58	0.58	0.58	0.58	0.58
TFC	0.52	0.13	0.15	0.15	0.38	0.35	0.35	0.37	0.38
$\boldsymbol{TFC}_{max}$	0.50	0.23	0.21	0.21	0.52	0.51	0.50	0.50	0.50
HFC	0.49	0.47	0.47	0.47	0.44	0.42	0.41	0.43	0.44

Therefore, f_a and f_b are both associated with Bug_1, so these features share that bug.

Using our feature coupling metrics and the defect data, we calculated the Spearman rank order correlation coefficient [210] to determine the relationship between the feature coupling measures and fault-proneness. Table 5.7 lists the Spearman correlation coefficients for dbViz and Rhino for all the versions of the corpora. Correlation coefficients can take values in the range of -1.0 to 1.0. A perfect negative correlation is denoted by -1.0, a perfect positive correlation is designated by a value of 1.0, and zero means no correlation. All of the Spearman correlations in Table 5.7 are statistically significant at the one percent confidence level, meaning there is only a 1% probability that the relationship is by chance.

The results for dbViz and Rhino indicate that there is a moderate to strong[10] correlation between the feature coupling metrics and defects. Under the default configuration (comments, no stemming) in dbViz, textual coupling had the strongest correlation (0.52) with bugs, while in Rhino structural coupling was the strongest (0.62). HFC is also moderately correlated with bugs in both systems. The correlations between bugs and the variants of our structural and textual coupling metrics, SFC' and TFC_{max}, are not very different from the metrics on which they are based. From these results, we can not support H_0, the null hypothesis, and can support H_1, the alternative hypothesis. In other words, *feature coupling is correlated with defects.*

Under the different versions of the corpus, SFC is unchanged since corpus building does not impact structural information. However, textual coupling does change, and with it, its correlation with bugs. In dbViz, TFC's correlation with defects is significantly impacted by the exclusion of comments and the use of stemming since dbViz is a relatively small system. TFC's correlation with bugs in Rhino does not suffer from the lack of comments or use of stemming as greatly as in dbViz, but there is still a slight weakening of the correlation. From this, we conclude that *the best configuration under which to build a corpus to measure textual feature coupling is to include comments but not to use stemming.* Stemming may be useful in other contexts [57], but we did not observe it to have an impact on these results.

Using this top-performing configuration, we created one additional corpus for Rhino that included the ECMAScript specification, an external document. By including this documentation in the corpus, we are adding domain information. The last column of Table 5.7 ($c - ns + d$) lists the correlation values between the metrics and bugs for this version of the corpus. The numbers in the table are rounded so it is not obvious, but for all the metrics except TFC_{max}, the version of the corpus with the strongest correlation with bugs is $c - ns + d$. Consequently, if programmers are seeking to use feature coupling to evaluate the fault-proneness of features and have documentation available, it should be included in the corpus for improved results. This finding supports other results in the literature that state that the inclusion of documents besides source code aide IR results [236].

[10]We use "strong" and "moderate" based on convention in [46], which have also been used in other software engineering contexts [63].

Table 5.8: Spearman correlation coefficients for HFC in dbViz and Rhino. All values are statistically significant at the one percent level (two-tailed). The sample size (number of feature pairs) is 78 for dbViz and 84,255 for Rhino.

	dbViz	**Rhino**		**dbViz**	**Rhino**
$\boldsymbol{S}_{0.05}\boldsymbol{T}_{0.95}$	0.52	0.38	$\boldsymbol{S}_{0.55}\boldsymbol{T}_{0.45}$	0.47	0.44
$\boldsymbol{S}_{0.1}\boldsymbol{T}_{0.9}$	0.53	0.39	$\boldsymbol{S}_{0.6}\boldsymbol{T}_{0.4}$	0.46	0.45
$\boldsymbol{S}_{0.15}\boldsymbol{T}_{0.85}$	0.53	0.39	$\boldsymbol{S}_{0.65}\boldsymbol{T}_{0.35}$	0.45	0.46
$\boldsymbol{S}_{0.2}\boldsymbol{T}_{0.8}$	0.52	0.4	$\boldsymbol{S}_{0.7}\boldsymbol{T}_{0.3}$	0.44	0.47
$\boldsymbol{S}_{0.25}\boldsymbol{T}_{0.75}$	0.52	0.41	$\boldsymbol{S}_{0.75}\boldsymbol{T}_{0.25}$	0.43	0.48
$\boldsymbol{S}_{0.3}\boldsymbol{T}_{0.7}$	0.51	0.41	$\boldsymbol{S}_{0.8}\boldsymbol{T}_{0.2}$	0.42	0.48
$\boldsymbol{S}_{0.35}\boldsymbol{T}_{0.65}$	0.51	0.42	$\boldsymbol{S}_{0.85}\boldsymbol{T}_{0.15}$	0.41	0.5
$\boldsymbol{S}_{0.4}\boldsymbol{T}_{0.6}$	0.51	0.42	$\boldsymbol{S}_{0.9}\boldsymbol{T}_{0.1}$	0.4	0.51
$\boldsymbol{S}_{0.45}\boldsymbol{T}_{0.55}$	0.5	0.43	$\boldsymbol{S}_{0.95}\boldsymbol{T}_{0.05}$	0.39	0.53
$\boldsymbol{S}_{0.5}\boldsymbol{T}_{0.5}$	0.49	0.44			

5.4.3.1 Hybrid Feature Coupling

In addition to investigating the five feature coupling metrics above, we also explored the effect of varying the weights assigned to our hybrid feature coupling metric, HFC. By varying the weights, preference is given to one type of information over the other, which may be useful in cases when one source of information is more reliable than the other. For instance, if a system is poorly structured but has good identifier names, more weight can be placed on textual coupling. Table 5.8 lists the Spearman correlation coefficients for all possible HFC combinations with a step size of 0.05 for the default corpus. All the correlations are statistically significant at the one percent confidence level. In dbViz, textual coupling is more strongly correlated with bugs than structural coupling (0.52 vs. 0.38), so increasing the textual weight improves HFC's correlation. The opposite is true in Rhino where structural coupling has a stronger correlation with bugs than textual coupling (0.62 vs. 0.38). Therefore, increasing the structural weight strengthens HFC's correlation with defects. Rhino may have a stronger structural coupling than dbViz since it is an order of magnitude larger in size. Overall, the HFC variants have moderate correlations with defects, and programmers using HFC should select weights based on their assessment of the system and type of coupling they want to emphasize. However, when the quality of the structured or unstructured information is unknown, using the default weight of 0.5 provides good results.

5.4.3.2 Comparison with an Existing Metric

The distance between features metric ($DIST$) introduced by Wong and Gokhale [233] is a feature metric that is very similar to coupling because it measures the distance (or similarity) between features. $DIST$ is computed based on information collected by dynamically executing a system. Since $DIST$ is the state of the art in feature measurement, we compared our metrics to it. $DIST$ was originally defined on basic blocks, but we redefine it here at the method level to be able to directly compare it with our metrics. Let M_a and M_b be the sets of methods executed by inputs that invoke features f_a and f_b respectively. Therefore, the distance between features f_a and f_b is

$$DIST(f_a, f_b) = \frac{M_a \oplus M_b}{M_a \cup M_b}. \tag{5.8}$$

where $\oplus$ is the exclusive OR operator.

We collected one execution trace for each of dbViz's 13 features and 51 of Rhino's. The dbViz traces were based on the developers' use cases, while the Rhino traces were based on available test cases, and not all features had a test case. We computed $DIST$ between all pairs of features and calculated the Spearman correlation to determine the relationship between $DIST$ and fault-proneness. Bugs were associated with features as described in Section 5.4.1. For dbViz, the Spearman correlation coefficient for $DIST$ and bugs is 0.02, and for Rhino, it is 0.05. Both values are not statistically significant. $DIST$'s correlation with defects is very close to zero, meaning that there is almost no correlation between the metric values and bugs. In comparison, *all of our metrics have positive moderate to strong statistically significant correlations with bugs.* $DIST$ is expensive to compute because of the overhead of collecting traces. It is not a good predictor of faults, likely due to the imprecise nature of dynamic analysis. In contrast, our metrics are less expensive, and all of them are good predictors of fault-proneness.

Besides being the only feature coupling metric with no statistically significant correlation to bugs, an example of $DIST$ highlights the problems associated with using dynamic information. Consider dbViz's features to start and to exit the system. The dynamic coupling between these two features is 1 because despite what other features are invoked, the

system must always be started and exited. This example shows the difficulty inherent in using dynamic information for feature coupling because some features cannot be invoked separately. On the other hand, SFC between these features is 0 and TFC is 0.08, reflecting the true lack of coupling between them, as is also supported by the fact that they do not share a bug.

5.4.3.3 The Confounding Effect of Size

Class coupling metrics have been shown to be good predictors of fault-proneness [9]. The higher coupling a class has, the more likely it is to have defects. However, larger classes are also more likely to contain defects, and class size has been shown to have a confounding effect on the association between coupling and fault-proneness [78] and change-proneness [245]. Without accounting for class size, the relationship between coupling and fault-proneness may be overestimated. Therefore, we must also investigate if size has a confounding effect on our feature coupling metrics.

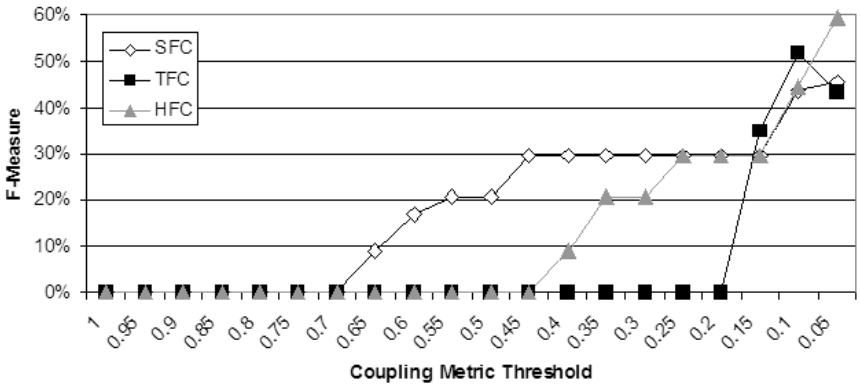
Traditional class coupling metrics are defined for a single class. For instance, CBO is the number of other classes that a class uses. By contract, our feature coupling metrics are defined for pairs of features and measures the degree of coupling among those two features. This presents a challenge when testing for the confounding effect of size. Instead of a single metric capturing how coupled a feature is, each feature has many different coupling measures, one for each other feature in the system. Also, instead of the size of a single feature, we have two features. The best we can do is determine if there is a correlation between the size of a feature (in terms of number of methods) and fault-proneness. The Spearman correlation coefficient between the number of methods associated with a feature and the number of bugs a feature has is 0.53 ($\alpha = 0.05$) in dbViz and 0.83 ($alpha < 0.001$) in Rhino. These results mean that there is a strong, statistically significant relationship between the size of a feature and the number of defects it has. This relationship could be confounding the correlation between the feature coupling metrics and bugs. However, SFC', the variant of SFC increases the size of a feature by considering the first order structural dependencies of a feature's relevant methods to also be relevant, and SFC' correlation with bugs is weaker than SFC's (See Table 5.7). Therefore, the confounding

effect of size on coupling metrics may not be as pronounced as it is for class coupling.

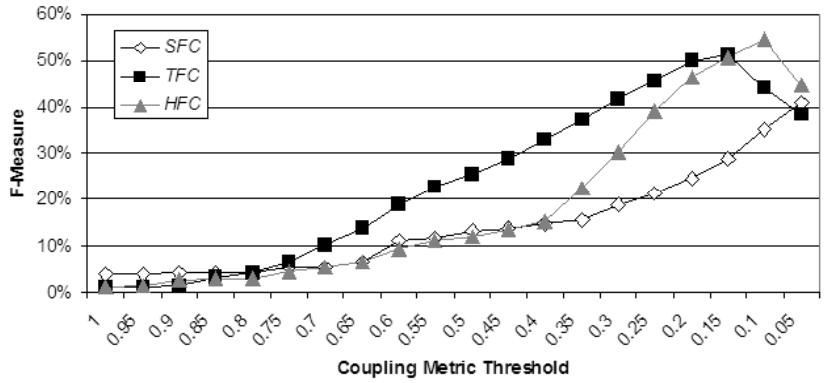
5.4.4 Using Structural and Textual Coupling to Support Feature-Level Impact Analysis

Our second case study investigates the application of feature coupling metrics for impact analysis. Given a starting point, such as a change to some module, impact analysis involves detecting other modules within a system that may be affected by a change [154, 171]. Both class-level coupling and information retrieval have been used for impact analysis [29, 164]. Generally, to select candidate modules to investigate for impact analysis, a threshold is set on the coupling or textual similarity values. Previous research on using coupling or information retrieval for impact analysis has focused on identifying methods and classes [164], not features. Therefore, we explore if feature coupling metrics can be used to find other features that are likely to be affected by a change to a feature undergoing modification by using defects identified in these features as an oracle.

To evaluate feature coupling in the context of impact analysis, we use available bug data from two systems to compute the precision, recall, and f-measure of the relevant coupled features recommended by our metrics. The process can be described as follows. For a bug b, we create a set $F_b = \{f_1, f_2, \ldots, f_n\}$ of features that all share the bug. That is, every feature in the set is associated with bug b. For each feature f_i in F_b, we determine which other features from all of the system's features are coupled to f_i by setting a threshold. For example, if the threshold is 0.5, then every feature that is coupled to f_i with a metric value equal to or above 0.5 is included in a new set T. Then, precision and recall are computed with T being the retrieved set and F_b (excluding f_i) as the relevant set. Precision is the ratio of the number of relevant features retrieved over the total number of features retrieved, while recall is computed as the number of relevant features retrieved divided by the total number of relevant features. The f-measure is the harmonic mean of precision and recall. For each bug b, we get precision, recall, and f-measure values. To get an overall measure of all bugs in the system, we summarize these precision, recall, and f-measure values using a macroevaluation averaging technique as in [247]. Macroevaluation means an average is taken of the values for all f_i in F_b and then for all bugs in the system. These values were



(a) dbViz.



(b) Rhino.

Figure 5.4: Average f-measure of coupled features for various thresholds.

computed for all threshold values with a step size of 0.05.

Figure 5.4 shows the average f-measure for dbViz and Rhino, and Table 5.9 shows the average precision and recall values of SFC, TFC, and one version of HFC ($S_{0.5}T_{0.5}$) at various coupling thresholds with a step size of 0.05 in Rhino. These results are for the default corpus. Focusing on the Rhino results, the best precision for structural coupling is 78.4% with a recall of 24.8% at a threshold of 0.1, while the best recall is 30.2% with a precision of 77.9% at the 0.05 threshold, meaning at best slightly over three quarters of the candidate features are relevant, but only 25% to 30% of the relevant features are found. Textual coupling's best performance in terms of precision is 54.4% with a recall of 38.1% at the 0.3 threshold, while its best recall of 86% with 28.1% precision is at a threshold of 0.05. The precision of SFC seems to increase and then level out as the threshold decreases. The precisions of both TFC and HFC increase until a certain point, then both decline,

Table 5.9: Precision and recall values for impact analysis of different metric thresholds in Rhino. The first value in a cell is precision, and the second is recall.

Threshold	SFC	TFC	HFC
1	40%, 3%	5%, 1%	5%, 1%
0.95	41%, 3%	5%, 1%	8%, 1%
0.9	43%, 3%	8%, 1%	18%, 2%
0.85	43%, 3%	17%, 2%	22%, 2%
0.8	45%, 3%	18%, 3%	25%, 2%
0.75	48%, 4%	24%, 5%	31%, 3%
0.7	50%, 4%	27%, 8%	35%, 4%
0.65	53%, 5%	30%, 11%	42%, 5%
0.6	58%, 7%	36%, 15%	48%, 6%
0.55	60%, 8%	37%, 19%	52%, 7%
0.5	66%, 9%	39%, 22%	57%, 8%
0.45	67%, 9%	42%, 25%	61%, 9%
0.4	71%, 10%	48%, 28%	66%, 10%
0.35	72%, 10%	52%, 33%	69%, 15%
0.3	75%, 13%	54%, 38%	68%, 22%
0.25	75%, 14%	53%, 45%	70%, 30%
0.2	77%, 17%	**52%, 54%**	68%, 39%
0.15	77%, 20%	46%, 65%	63%, 46%
0.1	**78%, 25%**	34%, 79%	**55%, 60%**
0.05	**78,% 30%**	**28%, 86%**	34%, 80%

likely due to the fact that the threshold is low enough that too many features are deemed textually coupled when they are not. SFC had the best precision overall but the worst recall. The precision for HFC generally fell below that of SFC but above TFC, and its recall is above TFC and below SFC. Therefore, using hybrid feature coupling is a good compromise between the two other metrics. For example, at a threshold of 0.1, HFC's precision is 55% and its recall is 60%. While feature coupling may not provide the best solution to the impact analysis problem, these results suggest that the metrics can still be useful. More research is needed to provide more practical techniques. However, these initial results are promising and comparable to some existing techniques on impact analysis based on structural and textual information [29, 164].

The precision and recall results also add weight to our claim that structural and textual feature coupling are complementary since their curves are different. We also executed the Kruskal-Wallis statistical test, a non-parametric alternative to the analysis of variance test, to assess if SFC and TFC are significantly different. At a significance level of 0.01, the test for both dbViz and Rhino show that SFC's and TFC's precision and recall values are

indeed significantly different.

The results are better if individual metric thresholds in Table 5.9 are considered. For instance, TFC at a threshold of 0.2 has 52% precision and 54% recall (f-measure: 50%), meaning one in every two features deemed coupled to a feature of interest would be impacted by a change. If these values are not high enough, a mix of metrics and thresholds could be used to achieve better results. For example, SFC has 78% precision at threshold 0.1, and it could be combined with TFC's recall of 75% at the same threshold.

The impact analysis results presented thus far have been for the default version of the corpus used to obtain textual information. We also investigate the use of feature coupling metrics for impact analysis using different corpora configurations (see Section 5.4.2) for the Rhino system. Figure 5.5 shows the average f-measure of TFC for the various versions of the corpus. Recall that only textual information is affected by the corpus' configuration, so SFC remains the same across corpora. The graph indicates that the way in which a corpus is built does little to influence precision and recall for impact analysis, no matter the threshold. However, the corpus with comments and stemming typically has the highest precision and recall. Just as was observed with the Spearman correlation coefficients, the inclusion of comments yields better results. However, textual feature coupling still works well in cases where comments are missing.

Finally, we study the effects of HFC's weights on precision, recall, and f-measure during feature level impact analysis. We provide only the results for the default corpora since the results for the other versions were similar. We select two metric thresholds that performed well for SFC and TFC (0.2 and 0.15) and calculate precision and recall of HFC for all weights with a step size of 0.05. Figure 5.6 shows the f-measure curves for HFC at a threshold of 0.2 (black lines) and 0.15 (gray lines). The x-axis denotes the structural weight. The corresponding textual weight is simply one minus the structural weight.

The graph illustrates the effect of relying on one type of information another. Depending more heavily on structural information yields good precision at the cost of poor recall. Overall, using HFC produces better results than the standalone SFC and TFC metrics. Consider HFC with a structural weight of 0.2 and a textual weight of 0.8 at a threshold of 0.2. The precision is 51% and the recall is 55%. At the same threshold, SFC's precision

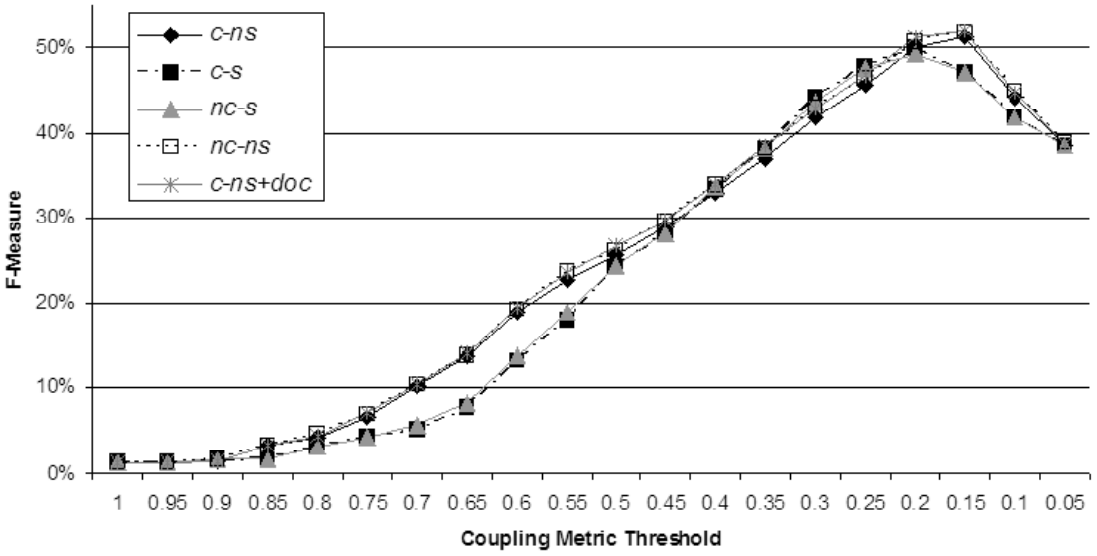


Figure 5.5: Impact analysis f-measure values of TFC for different Rhino corpora.

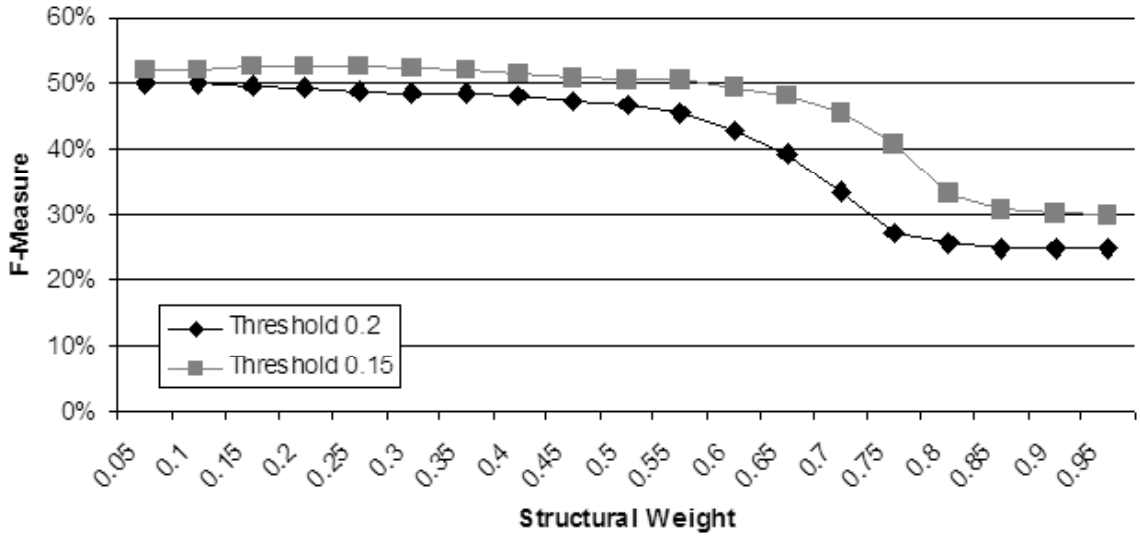


Figure 5.6: F-measure of HFC in Rhino at selected thresholds.

is 77%, but its recall is only 17%, so HFC is a better overall performer in this situation because its recall is much higher without sacrificing too much precision. Therefore, HFC helps alleviate those cases where the quality of either structural or textual information is low.

5.4.4.1 Feature-Class Coupling

There is a cost associated with computing feature coupling metrics: programmers must first identify the methods that implement a feature. They can use a feature location technique to do so. However, they may locate the implementations of only a subset of features in which they are interested instead of all of the features in the system. In this situation, programmers making changes to features may want to know which classes in the system may be affected by their changes. Therefore, we also investigated feature-class coupling. Instead of measuring the coupling between two features, these alternative metrics determine the degree of coupling between a feature and a class. Their definitions are similar to those of the feature coupling metrics given in Section 5.3, except instead of two features, one feature is replaced with a class and its methods.

In the same way that we explored if feature coupling could be used for impact analysis, we also explored if feature-class coupling could be used to determine classes that would be affected by a change to a feature. Figure 5.7 shows the average f-measure of using feature-class coupling in Rhino. The results are based on randomly selecting 40 of Rhino's features (about 10% of the total number of features) and considering only those features' implementations to have been located. The coupling between these features and Rhino's classes was computed. The same criteria as explained in Section 5.4.4 was used to determine bugs shared by features and classes. The results in Figure 5.7 are an average of 10 randomly selected set of features. In the best case, feature-class coupling has an f-measure of 22%. Comparatively, the best f-measure of the feature coupling metrics was above 50%. Feature-class coupling is not as effective at determining what would be affected by a change as the feature coupling metrics. The difference is likely due to the fact that not every method in a class would be impacted by a change, so looking at class coupling is more noisy. Features have scattered implementations, so metrics that are designed specifically to account for this

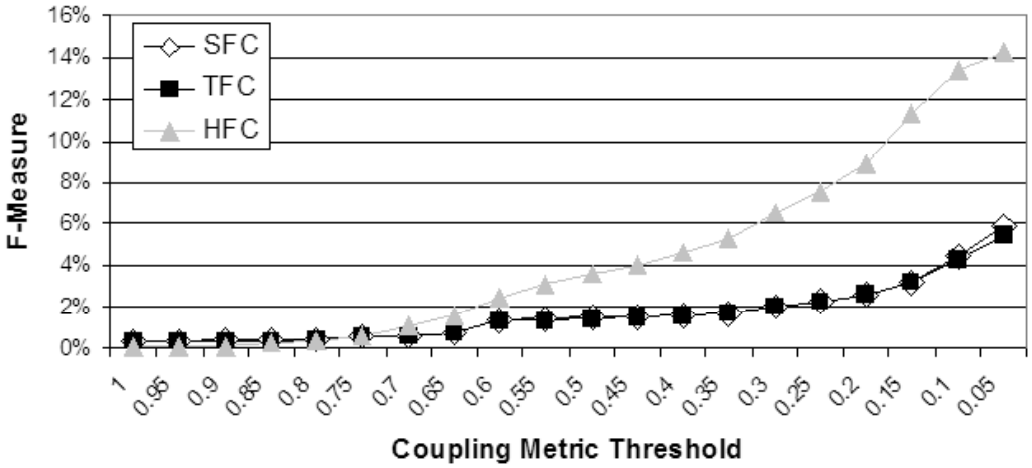


Figure 5.7: Average f-measure of feature-class coupling in Rhino.

dispersion are more effective.

5.4.5 Developer Study

In the final part of our evaluation, we investigate if our feature coupling metrics align with developers' opinions of feature coupling. If the metrics indicate that two features are coupled and so do the majority of developers surveyed, then we can be confident in the utility of the measures. More formally, we formulate two hypotheses.

$\boldsymbol{H}_2$ The null hypothesis is that there will be no consensus among the developers and the metrics about whether or not two features are coupled.

$\boldsymbol{H}_3$ The alternative hypothesis is that the majority of developers will indicate that two features are coupled when the features' SFC or TFC values are high and that the features are not coupled when either metric is low.

To test our hypotheses, we conduct a survey in which developers were asked to rate the strength of coupling among pairs of features. Below, we offer general details about the participants and the task they performed, as well as exploring the results of the survey.

5.4.5.1 Programmers

The respondents to our survey were 31 volunteer programmers from several different institutions. Twenty-three of the programmers were graduate students, one was an undergradu-

ate, and seven were industry professionals. On average, they had 7.2 years of programming experience, 3.8 with Java, and 2.6 with Eclipse. Each volunteer was given a link to the survey's instructions and could complete it on their own time. The survey took 97 minutes to complete, on average.

5.4.5.2 Task Description

The programmers downloaded an Eclipse installation that was preloaded with our FLAT^3 plug-in and all the necessary source code. FLAT^3 included mappings of features to code for selected features from Eaddy et al.'s [63] data sets. The programmers could click on a feature's name to see the methods associated with it and double click on a method to show its source code in the editor. The programmers were asked to consider the code of two features and rate whether the features were coupled. The responses varied according to a four-level Likert scale: "Strong No," "Weak No," "Weak Yes," or "Strong Yes." If a developer could not decide on a rating, they could respond "Unknown." The pairs of features included five from dbViz, six from Rhino, and five from iBatis. The exact instructions and pairs of features given to the participants can be found in Appendix C.

5.4.5.3 Agreement Among the Participants and with the Metrics

The survey is a rating of n subjects (the 16 feature pairs) by k raters (the 31 programmers). We tested if there was a sufficient amount of agreement among the developers' responses to be able to draw conclusions about the feature coupling metrics. To determine the amount of agreement among the raters, we designed our analysis in a fashion similar to [150] by using the intra-class correlation coefficient (ICC) [147]. We used $ICC(A,1)$, which calculates the agreement of all the raters, where each person rates each subject (feature pair). The A in $ICC(A,1)$ means it is an absolute agreement, and the one indicates the ratings are not an average. With the ratings stored in a matrix with feature pairs as the rows and raters as the columns, $ICC(A,1)$ is calculated as follows:

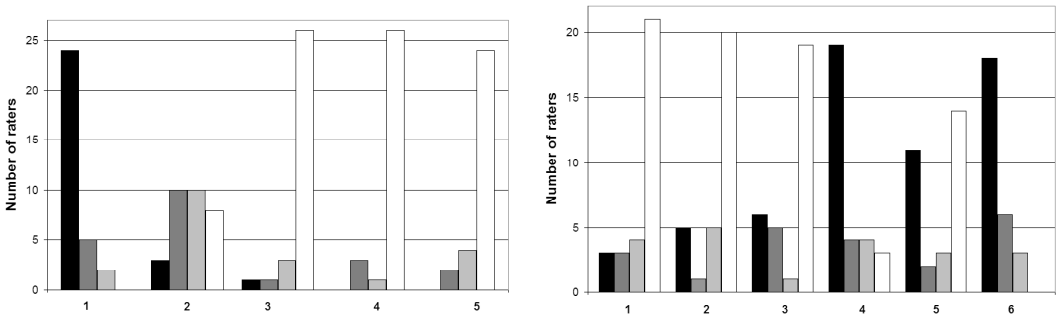
$$ICC(A,1) = \frac{MS_r - MS_c}{MS_r + (k-1)MS_e + k/n(MS_c - MS_e)} \tag{5.9}$$

where k is the number of raters, MS_r is the mean square for rows, MS_c is the mean square for columns, and MS_e is the mean square error. $ICC(A,1)$ relates the variance of the ratings of each feature pair to the overall variance.

The ratings given by the developers were ordinal, but numeric data is required to compute ICC. Therefore, we transformed the ratings of "Strong No," "Weak No," "Weak Yes," and "Strong Yes" to the values 1, 2, 3, and 4 respectively. Five programmers gave a rating of "Unknown" for at least one feature pair. These "Unknown" responses were omitted in the calculation of ICC. The $ICC(A,1)$ of the programmers in our survey is 0.45 (values can range from -1 to 1), meaning there is a moderate amount of agreement in their ratings of the pairs of features. We believe that there is enough concordance to be able to draw conclusions. The law of large numbers states that if the population sample is suitably large (between 30 to 50), then the central limit theorem applies even if the population is not normally distributed [208]. In our case, we have 31 participants, so the central limit theorem applies. While we had to remove some responses from the computation of ICC (replies of "Unknown"), the rest of our analyses are based on the responses of all 31 developers, so they can be considered significant.

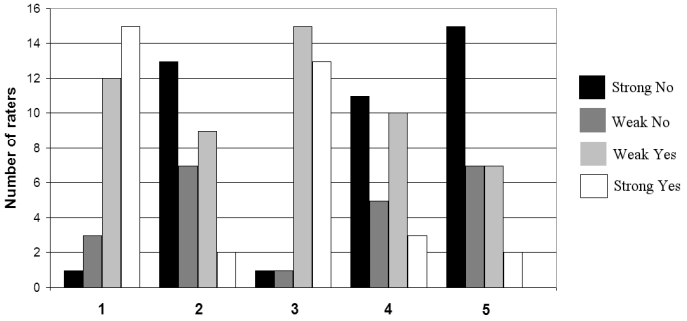
Knowing that the programmers have a sufficient amount of agreement about which feature pairs are coupled or not, we can examine if the developers' opinions support the feature coupling metrics. Figure 5.8(a), Figure 5.8(b), and Figure 5.8(c) summarize the number of developers that gave each rating for dbViz's, Rhino's, and iBatis' feature pairs, respectively. The height of each bar corresponds to the number of developers that gave the feature pair that rating. Note that each feature pair may not have the same number of total ratings because responses of "Unknown" are excluded. Each cluster of bars can be compared to the metric values in Table 5.10. For instance, the first group of bars in Figure 5.8(a) corresponds to the pair dbViz #1, "Connect to database" and "Exit dbViz."

When both SFC and TFC are low, the majority of responses are "Strong No," as can be seen by the first pair of features in dbViz. These features are to connect to a database and exit the program and have little in common, so low structural and textual coupling values are valid, as supported by the developers' ratings. Additionally, these features do not share any common bugs, which is further evidence that they are not coupled.



(a) dbViz feature pairs.

(b) Rhino feature pairs.



(c) iBatis feature pairs.

Figure 5.8: Number of each rating for dbViz's, Rhino's, and iBatis' feature pairs.

Table 5.10: Feature coupling values for the dbViz, Rhino, and iBatis feature pairs in the developer survey. Bug data was not available for iBatis.

Features	**Pair**	***SFC***	***TFC***	**Bugs**
Connect to database & Exit dbViz	dbViz #1	0	0.03	0
Autoarrange Diagram & Undo/Redo	dbViz #2	0.07	0.06	0
Import from database & Import from SQL	dbViz #3	0.61	0.15	1
Add table & Remove table	dbViz #4	0.85	0.22	0
Save/Load diagram & Load saved diagram	dbViz #5	0.45	0.13	2
Unary + operator & Addition operator	Rhino #1	0.33	0.27	15
Addition operator & Subtraction operator	Rhino #2	0.71	0.28	17
Date.prototype.toString & Date.prototype.valueOf	Rhino #3	0.75	0.74	2
Unicode format chars & ToPrimitive	Rhino #4	0	0.08	0
parseInt & parseFloat	Rhino #5	0.4	0.46	2
SQRT2 & Date.prototype.getTimezoneOffset	Rhino #6	0	0.85	1
Data Sources & JTA	iBatis #1	0.08	0.42	N/A
JBDC & JTA	iBatis #2	0	0.44	N/A
Query & Max Results	iBatis #3	0.15	0.47	N/A
Update & Autogenerated Keys	iBatis #4	0	0	N/A
SELECT & SQL Scripts	iBatis #5	0	0.06	N/A

Another overall trend is that when SFC is high, most raters responded "Strong Yes." As an example of high structural coupling, consider the feature pair dbViz #3, "Import from database" and "Import from SQL file." These features are very similar in function and share a number of methods, so a high SFC value (0.61) makes sense, and the programmers' responses also support SFC being high. Furthermore, the two import features have a common bug, which also supports higher coupling between them. However, TFC between these two features is rather low (0.15) because the methods that are distinct to each feature have their own vocabulary. Since the metrics are low when most participants responded "Strong No" and high when the responses were "Strong Yes," we can reject H_2, the null hypothesis, and support H_3, the alternative hypothesis.

One interesting case is the Rhino #6 feature pair. The two features are "SQRT2," the number value of the square root of two, and "Date.prototype.getTimezoneOffset" that gets the local time and UTC in minutes. There is no structural coupling between the features, but rather high textual coupling. The majority of responses for this feature pair were "Strong No" despite these two features having a high TFC value. Two options are possible: the features are not actually coupled and the high textual coupling is a coincidence, or the programmers did not pick up on the similarity in the two features' vocabularies because textual coupling is not as well known a concept as structural coupling. The two features do have a shared bug, but after reviewing the features' source code, the textual coupling seems to be artificial. "SQRT2" has two methods, and both of those methods' names happen to be the same as two of "Date.prototype.getTimezoneOffset" three methods. These methods perform similar parsing functionalities and use many of the same variable names, so high textual coupling in this case seems to be accidental.

Another interesting case is the Rhino #5 feature pair: "parseInt" and "parseFloat." SFC and TFC have approximately equal values which are both substantially greater than the average for each metric in Rhino. The developers are almost evenly split in their opinions of whether these two features are coupled, with a slight majority thinking they are coupled. The feature's coupling is also supported by the fact that they have two bugs in common. The developers' mixed ratings suggest that perhaps there is a coupling threshold, but that threshold varies from person to person.

Overall, the ratings given by the programmers seem to support our feature coupling metrics. This implies that the measures do capture the coupling between features. Generally, the respondents' opinions support SFC more than TFC, but that may be due to the fact that textual coupling is a newer concept.

5.4.6 Threats to Validity

In this section, we discuss the main threats to the validity of our case studies and provide details on how we minimized these threats.

5.4.6.1 Internal Threats to Validity

Internal validity refers to the degree to which statements about cause and effect are valid. Since we use previously published data sets, we inherit all of the threats to validity associated with them. One internal threat of the data sets is the subjective manner in which methods were assigned to features. These facts limit the consistency of our results because different mappings would produce different results. However, since the data sets have been used and verified by other researchers [62, 63], these threats are minimized. Additionally, Spearman rank-order correlation can mitigate unreliable measurements as long as their relative order is correct [110]. Also, the Rhino data set has a large sample size (84,252 feature pairs). The moderate and strong correlations observed are unlikely if the data is unreliable. Another threat we inherit from the data sets pertains to the assignment of bugs. As with any approach to mining software repositories, defects can potentially be mapped to wrong or missing methods if methods undergo a change in signature. Similarly, automated repository mining does not always provide a complete picture of a bug's history. It may lack social, technological, and organizational knowledge [6] or may be biased and only record a fraction of bug fixes [14].

Another threat related to the data sets is their granularity. Full methods are associated with features. However, only a small portion of the code in a method may actually pertain to a feature [118, 172]. Therefore, a finer level of granularity such as statements or basic blocks would be more accurate. Since we are not experts in any of the systems we studied, we made no attempts to refine the granularity of the data sets.

In our case studies, we observed a high correlation between feature coupling and defects, which may imply that feature coupling can serve as a predictor for faults. However, correlation values only measure goodness of fit, not predictive power. To better assess predictive power, we would need to perform some form of data splitting, such as ten-fold cross-fold validation, which is part of our future work.

In the context of our survey, there are a number of threats to validity. First, the programmers' proficiency with Java and Eclipse is a threat because we did not select participants based on their familiarity with either technology. Some of the programmers had no experience with Java or Eclipse. By including programmers with little or no Java experience in our survey, there is a danger that they made poor choices due to their unfamiliarity with the programming language. Another threat related to the programmers is their motivation. All the developers who participated were volunteers and received no compensation, so there was no motivation for them to perform well. On the other hand, there was no time pressure to complete the survey quickly because there was no time constraint.

Two final threats to the validity of our survey pertain to the task the programmers were asked to complete. The participants were instructed to consider if two features were coupled in the context of performing a change task to either, leaving the task rather open-ended and general. However, it may be difficult to gauge the relationship between two features without a specific context. There could be changes made to a feature that affect the other one, but other changes made to the same feature may not affect the other feature. To avoid making a judgment about a specific change task, we kept the task general.

5.4.6.2 External Threats to Validity

External threats to validity limit the degree to which generalizations can be drawn from our results. We studied only three systems, one small and the other two medium in size. In future work, our feature coupling metrics will be validated on larger systems. However, the number of features studied in Rhino was large (411), and the feature coupling metrics of both systems had statistically significant correlations with bugs. While the systems are open-source, their development shares many characteristics in common with industrial systems such as the use of specifications, use cases, and change management systems.

Therefore, it is reasonable to expect that our results would hold for industrial software of similar sizes. All the systems we studied are written in Java. To see if our results are not language-specific, additional studies on systems written in other languages are needed.

Concerning the survey we conducted, there are also threats to external validity. The majority of the programmers were graduate students, so the participants are not necessarily representative of all developers. However, some of our participants were industrial programmers, and in general, their responses aligned with those of the graduate students.

5.5 Conclusion

We have introduced novel metrics that capture feature-level coupling by using structural and textual information, filling a critical gap in the area of software measurement. We have theoretically validated our metrics and extended the unified framework for coupling measurement [25] with important new dimensions. Through our three-pronged evaluation, we have shown that these metrics are useful because they are good predictors of fault-proneness. Additionally, they have an application in feature-level impact analysis to determine if a change made to one feature may have undesirable effects on other features. Finally, based on the results of a survey of 31 developers asked to rate the strength of coupling between pairs of features, our metrics align with those ratings. Altogether, these results point to a solid conclusion that structural and textual feature coupling metrics are valid and useful tools for developers performing feature-level software maintenance.

A secondary goal of this work was to discover the optimal way in which to obtain our metrics so developers can use them most efficiently. Both TFC and HFC can be computed under different configurations. Textual information can be mined based on several options (i.e., include comments, perform stemming). When available, external documentation should be included in the corpus to boost textual similarities by adding more domain terminology and concepts. In the absence of external documentation, comments should be preserved. When combining structural and textual information for HFC, more weight should be placed on the stronger of the two sources to be able to better predict faults or perform impact analysis tasks. If the quality of the structural and textual information is

unknown, placing equal weight on each still performs well.

We make all of the source code, data, and results of the case studies available and invite other researchers to replicate our work.

Chapter 6

FLAT^3: Feature Location and Textual Tracing Tool

During software maintenance, it is very common for developers to search for source code that is relevant to their task. When their task pertains to modifying, extending, or adding functionality, their search is known as *feature* (or *concept*) *location* [5, 12]. For example, assume a developer working on an open source text editor needs to modify the *file saving* feature. The developer first needs to find the existing source code that implements file saving before she can make any changes. If the developer has never worked with this particular feature before, she will not know where to begin and may spend significant time and effort manually searching for the feature's source code before being able to make any changes.

To aid developers in this situation, automated feature location techniques have been proposed to reduce the amount of time and effort spent searching for a feature's implementation. Some of these approaches employ information retrieval (IR) to search a body of text, such as source code, for sections that are relevant [142]. Other techniques analyze dynamically-collected execution traces to identify a feature's implementation [76, 229]. IR and dynamic analysis have also been combined to form hybrid feature location techniques [5, 130].

To make these feature location approaches more accessible to developers, we have cre-

ated FLAT^3, the **F**eature **L**ocation and **T**extual **T**racing **T**ool[1]. It is an Eclipse plug-in that supports three well-established feature location techniques[2]: 1) information retrieval (IR), 2) dynamic collection of execution traces, and 3) a combination of IR and dynamic tracing known as SITIR [130]. Feature location via IR involves textually searching a project's source for code that is similar to a query that describes a feature. Dynamic feature location entails running the software and invoking the feature of interest to capture a trace of the source code that was executed. FLAT^3 also implements SITIR, which integrates textual and dynamic feature location techniques so that they can be used together effectively.

In addition to providing support for multiple feature location techniques, FLAT^3 also supports annotating and saving relevant search results. The tool permits developers to create and name features to which the source code implementing them can be linked. This feature mapping functionality allows developers to save their feature location results and avoids the need to repeatedly search for a given feature's implementation. It also allows developers to automatically compute feature coupling metrics.

FLAT^3 makes two significant contributions that current feature location tools do not provide. First, existing tools generally support one way of searching (i.e., IR only or dynamic tracing only). FLAT^3 makes both the IR and dynamic techniques available, and it also integrates them. FLAT^3's second contribution is its feature annotation function which documents a feature's source code and can be used to compute metrics about features. While there are some tools that provide this tagging functionality [181, 184], they are not coupled with feature location techniques, and existing feature location tools do not provide mechanisms for saving the mappings of features to source code. FLAT^3 is a complete suite of feature location, annotation, and visualization tools.

6.1 FLAT^3

FLAT^3 is implemented as an Eclipse plug-in. Figure 6.1 gives an overview of FLAT^3's architecture. The tool combines the functionality of several existing libraries and appli-

[1] FLAT^3 is available online at `http://www.cs.wm.edu/semeru/flat3/`.

[2] Part of the future work planned for FLAT^3 is to have it implement the feature location techniques based on web mining introduced in Chapter 4.

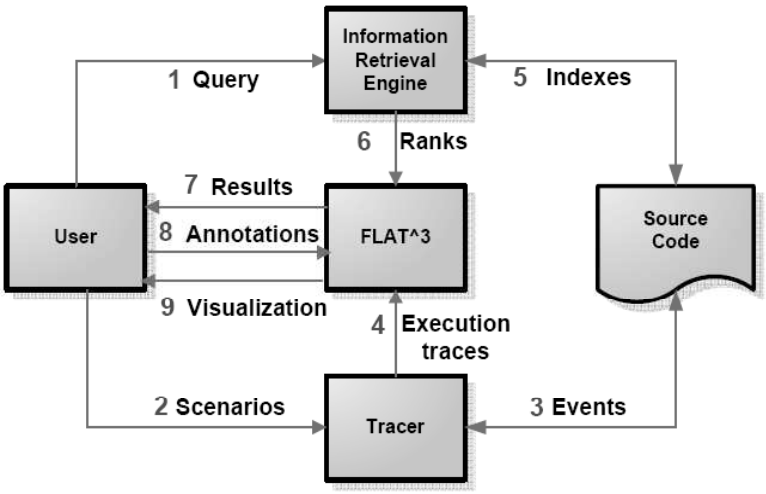


Figure 6.1: Overview of the architecture of FLAT^3.

cations. It uses information retrieval from the Lucene[3] library to locate and rank code by similarity to a user's query. FLAT^3 also uses MUTT[4] to capture execution traces of feature-specific scenarios and test cases. FLAT^3's feature annotation capability is based on ConcernMapper[5] and ConcernTagger[6], Eclipse plug-ins that allow for the creation of concern (feature) models and for source code to be linked to features. By integrating these existing tools, FLAT^3 provides developers with a way to easily search for features' implementations and annotate their findings for future reuse. Based on the annotations, FLAT^3 can also visualize the location of a feature's source code across a system's classes using a map metaphor similar to the one used in AspectBrowser[7]. FLAT^3's features are described in detail below.

6.1.1 Textual Feature Location

The first way in which FLAT^3 allows developers to perform feature location is textually. FLAT^3 textually searches for a feature's source code by leveraging the Lucene information retrieval library. To use this functionality, developers open the FLAT^3 Features view in Eclipse and click on the search toolbar button. This action opens a dialog box (See Figure

[3] `http://lucene.apache.org/java/docs/`
[4] `http://sourceforge.net/projects/muttracer/`
[5] `http://www.cs.mcgill.ca/~martin/cm/`
[6] `http://www1.cs.columbia.edu/~eaddy/concerntagger/`
[7] `http://cseweb.ucsd.edu/~wgg/Software/AB/`

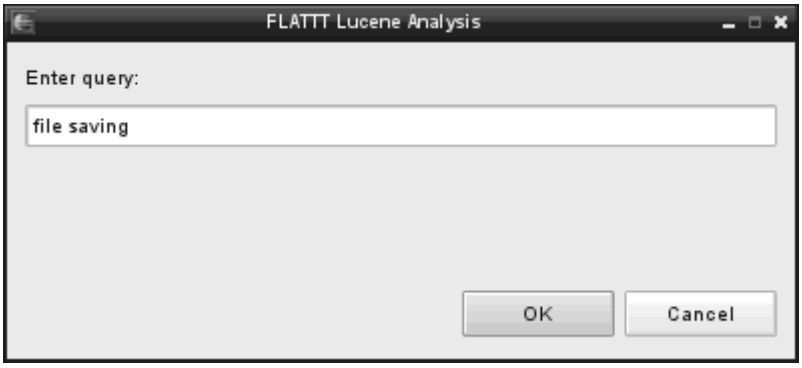


Figure 6.2: Entering a query to begin textual feature location.

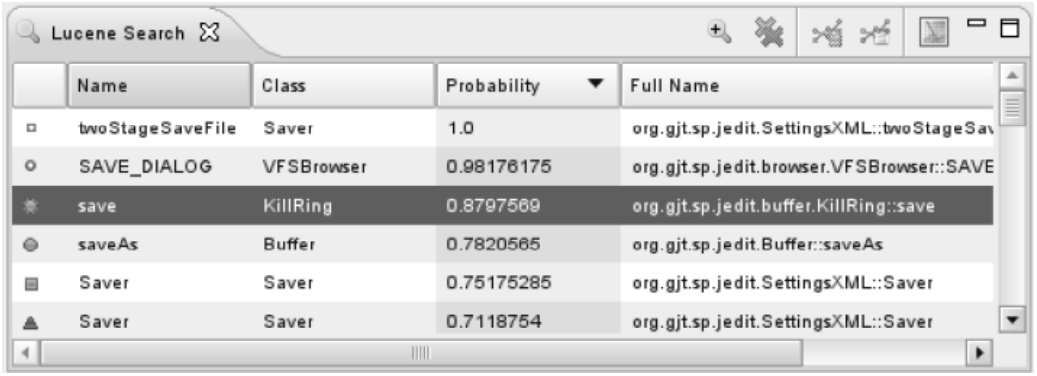


Figure 6.3: FLAT^3's Search/Trace Results view with a list of classes, methods, and fields returned by Lucene sorted by similarity to a query.

6.2) into which developers can enter a query that describes the feature they are trying to find, such as "file saving." After the query is issued, Lucene indexes Eclipse's workspace if it has not already been indexed. Indexing involves creating a document for each method and field consisting of all the words used in the method or field. Keywords and common stop words (e.g., "the" and "a") are removed. Also, words are split (e.g., "compoundIdentifier" becomes "compound" and "identifier") and stemmed (e.g., "searching" becomes "search"). Each document is converted to a vector, as is the query. Then, all the document vectors are compared to the query vector to determine their similarity, and a score is assigned to each method or field based on that similarity.

Figure 6.3 shows FLAT^3's Search/Trace Results view, listing the results returned by Lucene for the "file saving" query from the source code of jEdit, an open source text editor. The results include the method or field's name, class, a score of how similar it is to the query, it's fully qualified name, and any features with which it has been previously

annotated (not visible in the figure). The results are ordered by their relevance to the query. Developers can double click on a result to view that method or field's source code. If a result is deemed to be relevant to the feature of interest, it can be annotated in this view, as will be explained in Section 6.1.4. Developers can also refine their results by searching within the original results with a new query.

6.1.2 Dynamic Feature Location

In addition to textual feature location, developers can also use FLAT^3 to locate features dynamically. This approach to feature location uses MUTT, a tracing tool based on the Java Platform Debugger Architecture[8] (JPDA). MUTT runs a subject program on its own Java virtual machine and collects a trace of runtime method calls. What is unique about MUTT is the user can control when to turn tracing on and off with a button.

To perform dynamic feature location in FLAT^3, developers first determine a scenario or test case that invokes the desired feature. For instance for the *file saving* feature, a scenario would be to start jEdit, open a file, make changes, save the file, and exit. To collect an execution trace, developers right click on the class that contains the project's main method and select "Trace with MUTT," as in the first part of Figure 6.4. This will launch the program along with a separate window with a start/stop button to control tracing, as in the second part of Figure 6.4. The start button should be clicked just before the feature is invoked, and tracing should be stopped just after the feature's behavior completes. All methods that were executed between the start and stop interval are collected in a trace. Once developers are done tracing, they can close the application and return to FLAT^3 to find a listing of the methods executed by the scenario. The listing is very similar to Lucene's results (see Figure 6.3) with the exception that no similarity scores are given. Developers can browse these results to find relevant methods instead of searching the full source code of the system. Just as with Lucene's results, double clicking a method from the trace opens its source code for viewing. Traces can be saved and loaded again instead of having to be recollected, as shown in Figure 6.5.

[8] `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`

Figure 6.4: Invoking MUTT on jEdit in Eclipse (1) and jEdit running with MUTT's tracing control button (2).

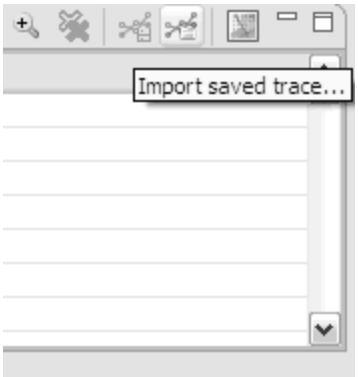


Figure 6.5: The Feature view's toolbar buttons to refine a search, return to the original search, export a trace, import a trace, and visualize a feature or search results.

6.1.3 Integrated Feature Location

FLAT^3 allows for the integration of its two separate feature location techniques. Since dynamic feature location in FLAT^3 is likely to return many methods, to narrow the results, it can be integrated with textual feature location following the SITIR approach [130]. After collecting an execution trace for a feature, IR is used to rank only the invoked methods instead of all of the methods in the system. In FLAT^3, after collecting a trace with MUTT, Lucene can be used to textually search only within the executed methods by clicking the "Refine Search" button. This opens a dialog in which developers can enter a query, causing Lucene to compute similarity scores for the methods in the trace as described in Section 6.1.1. The methods are indexed beforehand, and only similarities are computed at this point. After the scores are calculated, developers are presented with a list of the trace's methods ranked by their similarity to the query. Combining two types of feature location techniques employs more sources of information to find a feature's implementation than a standalone approach. Dynamic tracing acts as a filter to IR by limiting the methods that are ranked to only those that are executed. This idea was first introduced in the PROMESIR approach [160] and further refined in SITIR [130].

6.1.4 Annotating Features

Once a feature's source code has been found, it can be annotated and saved with FLAT^3. In the Features view, features can be created and given a name. Then classes, methods, and fields can be associated with a feature from any of the results views by right clicking on the method and selecting "Link" and the name of the feature to which the code belongs, as in Figure 6.6. Code can also be mapped to features through Eclipse's package explorer, outline view, and editor. Code can be mapped to multiple features. Once code has been linked to a feature, the feature's name appears in the search results, as in Figure 6.7 where the method "Saver.Saver" has been tagged with the *File Saving* feature.

Figure 6.8 shows the Features view, listing the code associated with the *File Saving* feature. A feature's methods are grouped hierarchically by class. FLAT^3 also supports a hierarchy of features so that a feature can have child features, as shown in Figure 6.8,

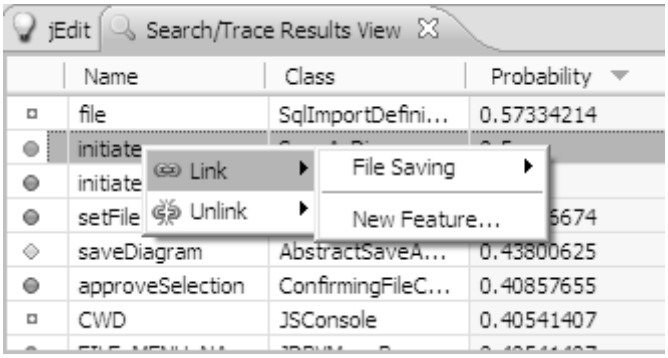


Figure 6.6: Linking a search result with a feature.

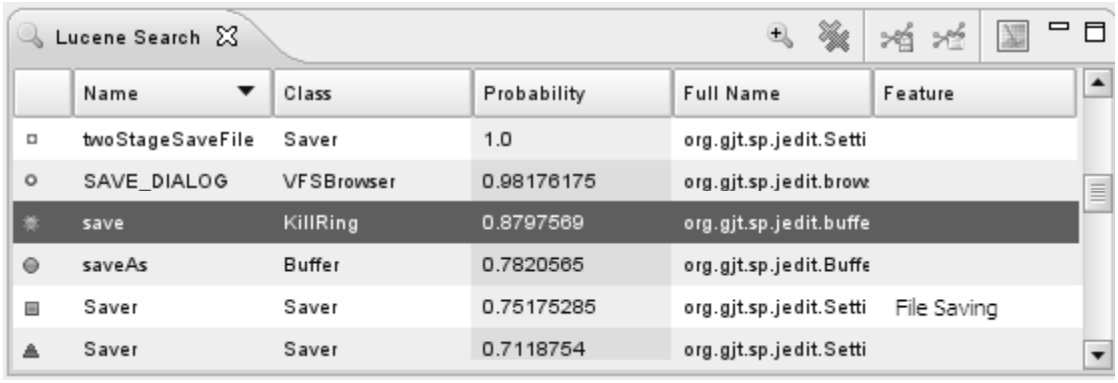


Figure 6.7: A search result has been annotated with the *File Saving* feature.

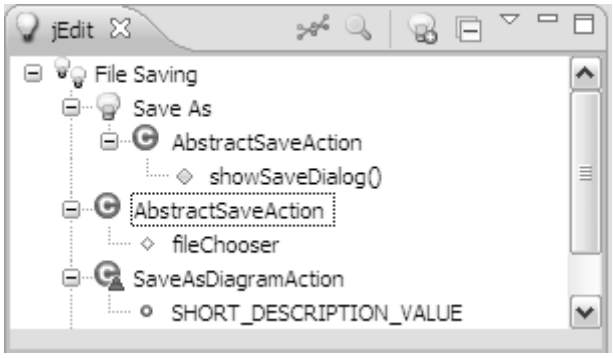


Figure 6.8: FLAT^3's Features view showing code associated with the *File Saving* feature. FLAT^3 supports child features. Here, the *File Saving* feature with a child feature, *Save As*.

Figure 6.9: FLAT^3's visualization view showing classes from a search that have code similar to the query. The highlighted rows indicates the degree of similarity of the code to the feature.

where *Save As* is a child of the *File Saving* feature. Code can be removed from features by right clicking on them and selecting "Unlink" and the name of the feature. Features and their mappings are saved and can be revisited when FLAT^3 is reopened. FLAT^3 supports multiple feature domains so that features from different software systems can be kept separate. Saving the mappings of source code to features acts as a form of documentation, making it easier to keep track of and modify features and their implementations [182]. An additional benefit of annotating a feature's source code is that FLAT^3 can automatically compute the feature coupling metrics defined in Chapter 5.

6.1.5 Visualization

FLAT^3 also provides a visualization functionality that shows the distribution of a feature or search results across files. The visualization is accessible by right clicking on a feature and selecting "Visualize feature..." or by clicking the "Visualize" button after obtaining results from Lucene or MUTT. FLAT^3 uses the same map metaphor as AspectBrowser [96] to visualize the location of aspects in files. Figure 6.9 shows an example of the FLAT^3 visualization. Each box represents a class, and each row of pixels in a class' box corresponds to a section of code. If the row is highlighted in red, it means that code is associated with the feature or present in the search results. If Lucene's results are visualized, the shade of the row of pixels indicates the degree of similarity of that section of code to the user's

query. This visualization gives developers a global idea of where a feature of interest is implemented.

6.2 Related Work

FLAT^3 is based on several existing tools. The Lucene library provides full-text searches, MUTT collects execution traces, and ConcernTagger and ConcernMapper [184] lend the ability to annotate and save feature mappings. These functionalities are integrated in FLAT^3. There are other existing tools that implement either feature location or annotations, but not both. IRiSS [163], JIRiSS [162], and Google Eclipse Search [166] are tools that support feature location via Latent Semantic Indexing (LSI) [59], an advanced IR method. FLAT^3 relies on Lucene, so it is faster than LSI-based tools. While none of these tools allow for the saving of located feature code, FEAT [181] and ConcernTagger do. However, these tools rely on manual feature location. There are several other feature location tools such as STRADA [69] which uses dynamic information; JRipples [35] and Suade [224] which use static analysis; Find-Concept [201] which uses natural language processing; and Dora [102] which uses textual and static analysis. However, FLAT^3 is unique in that it combines textual and dynamic feature location with annotations and visualization.

6.3 Conclusion

FLAT^3 is a novel tool suite for feature location and feature coupling. It is implemented as an Eclipse plug-in and combines the functionality of a number of existing tools in one easy-to-use application. FLAT^3 allows developers to perform feature location textually and dynamically, to save their results for future reference, to visualize the dispersion of features or search results throughout a project, and automatically compute feature coupling metrics. Future work on FLAT^3 includes incorporating feature location techniques based on web mining introduced in Chapter 4, making it more robust to be able to index large source code bases, trace larger programs, and save and update annotations for evolving programs.

Chapter 7

Conclusion

The motivation of this dissertation is to support feature-level software maintenance. Most maintenance tasks (e.g., bug reports) are expressed in terms of features, so supporting maintenance at the feature-level is more user-oriented than traditional class-level approaches. Features also tend to have non-modularized implementations, meaning that locating them and determining the relationships between them is difficult. This dissertation has introduced novel methods for supporting two software maintenance tasks: feature location and impact analysis via feature coupling.

Regarding feature location, this dissertation makes a number of contributions:

- We have conducted a survey of published research articles related to feature location. The articles have been classified within a taxonomy that has nine dimensions. Each dimension captures an essential characteristic of feature location research. The survey can be used by both researchers and practitioners to discover useful approaches and potential avenues for future research.
- We have completed an exploratory study of existing feature location techniques with the goal of determining how well the approaches locate multiple methods that are relevant to a feature. This study examined techniques that employ textual, dynamic, and static analyses. We explored different combinations of these analyses and different configurations of each. We found that existing feature location techniques can successfully locate one relevant method for a feature but rarely many more. These results led us to focus on developing new, more effective feature location techniques.

- We have introduced new feature location techniques that employ web mining algorithms to rank the methods executed in a trace and then use that ranking to filter false positives from the results of an IR-based approach. Our evaluation shows that pruning the bottom-ranked methods according to the HITS hubs algorithm is the most effective approach. Statistical analysis also shows that the improvement in effectiveness of our web mining approaches over the baseline is significant.
- We have developed tool support for feature location. The tool, called FLAT^3, allows users to search for features textually and dynamically, annotate the results, and visualize their annotations.

This dissertation also makes a number of contributions related to feature coupling:

- We have developed feature coupling metrics based on structural information, textual information, and their combination. All existing coupling metrics are defined at the class-level, so our metrics are novel and fill a gap in the research area.
- We have shown that there is a moderate to strong statistically significant correlation between our feature coupling metrics and defects. Just like with classes, the more coupled two features are, the more likely they are to have defects.
- The feature coupling metrics can be used for impact analysis. If a modification is made to one feature, the metrics can be used to determine what other features may be affected. In our evaluation, we found that as many as half of the features deemed as coupled would be affected by a change to the given feature, and over half of the affected features are retrieved.
- We conducted a survey with 31 programmers to determine if the feature coupling metrics indeed capture a recognizable relationship between features. The programmers were asked to rate the strength of coupling between 16 pairs of features, and the results show that when the programmers rated the features as tightly coupled, the metrics' values were high, indicating stronger coupling. Likewise, when the programmers rated the features as loosely coupled, the metrics' values were low, indicating an absence of coupling.
- We have developed tool support for our metrics. The FLAT^3 also supports the automatic computation of our feature coupling metrics from the annotations provided

by the user.

In addition, all of the data generated in this dissertation is made publically available online so that other researchers can replicate this work.

While the work presented in this dissertation shows promising results for the new feature location techniques and feature coupling metrics, there is still room for improvement in future work. In terms of feature location, there are several possible avenues. We have seen in two separate studies that feature location techniques do very well at locating one method that is relevant to a feature. However, features are often implemented by multiple methods, so approaches that more effectively locate more of a feature's source code are needed. Additionally, since the feature location techniques we have introduced make use of thresholds, an exploration of how to automatically select a threshold for a given feature is an area of future work.

There are also a number of avenues for future work related to feature coupling. We can expand the metrics' definitions to include fields or to be more fine-grained than methods. We have shown that there is a statistically significant correlation between feature coupling and defects. However, correlation measures goodness of fit and not predictive power. Future work includes performing data splitting to assess the predictive power of the metrics. The metrics also need to be computed for more features in other types of software systems to determine their generalizability.

In conclusion, this dissertation has introduced novel feature location techniques and feature coupling metrics to aid programmers performing software maintenance on features. We have shown these techniques and metrics to be effective and useful and envision that one day they may be widely adopted by software developers and maintainers.

Appendix A

Classification of Feature Location Articles

This appendix contains tables listing 1) the dimensions of the feature location taxonomy and their related attributes and 2) the surveyed papers classified within the taxonomy.

A.1 Dimensions of the Feature Location Taxonomy

The feature location taxonomy has nine dimension. Section 2.1 of Chapter 2 discusses these dimensions, and Table A.1 lists each dimension and its associated attributes.

Table A.1: Dimensions of the feature location taxonomy.

Dimension	Attribute	Description
Type of Article	Research (R)	The article introduces a new feature location technique.
	Tool (T)	The article describes a tool that supports feature location.
	Case Study (CS)	The article presents a case, industrial, or user study.
Type of Analysis	Dynamic (D)	Dynamic analysis is used to locate features.
	Static (S)	Static analysis is used to locate features.
	Textual (T)	Textual analysis is used to locate features.
	Historical (H)	Repository mining is used to locate features.
	Other (O)	Another type of analysis is used to locate features.
Source of Information	Source Code (SC)	Source code is used to locate features.
	Documentation (Doc)	Documentation is leveraged to find features.
	Execution Trace (ET)	Execution information is used to locate features.
	Dependence Graph (DG)	Features are found using a dependency graph.
	Repository (Rep)	Features are located by mining a repository.
	Other	Another source of information is used for feature location.
Granularity	Class (C)	The classes related to a feature are found.
	Method (M)	The methods that implement a feature are identified.
	Statement (St)	Statements, lines, or basic blocks associated with a feature are located.
	Variable (V)	Variables relevant to a feature are located.
	Artifact (A)	Non-code artifacts are associated with features.

Table A.1: (continued).

Dimension	Attribute	Description
Programming	Java	The approach supports feature location in Java.
Language	C/C++	The technique can find features for C/C++ systems.
Support	Other	Feature location in some other language is supported.
Presentation	Ranked List (Ranked)	The located program elements are presented as a ranked list.
of Results	Suggestion Set (Set)	The found program elements are presented as an unordered set.
	Visualization (Visual)	The located program elements are presented using a visualization.
	Manual	The feature location technique is a manual process.
Evaluation	Preliminary (P)	The evaluation is on small systems or preliminary evidence is given.
	Benchmark (B)	The evaluation is based on a predetermined benchmark.
	Expert (E)	System experts are used to evaluate the results.
	Non-expert (NE)	Non-experts are used to evaluate the results.
Comparison	ASDG	Abstract System Dependence Graph [39]
	DFT	Dynamic Feature Traces [77]
	FCA	Formal Concept Analysis-based feature location [76]
	grep	UNIX grep
	LSI	Latent Semantic Indexing-based feature location [142]
	PROMESIR	Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval [160]
	SPR	Scenario-based Probabilistic Ranking [5]

Table A.1: (continued).

Dimension	Attribute	Description
Comparison	SR	Software Reconnaissance [229]
(continued)	Suade	Suade Topology-based Search Tool [224]
	Other	Comparison to another feature location technique.
	None	No comparison to any feature location approach.
Systems evaluated		The software systems upon which the technique has been applied are listed.

A.2 Classification of Surveyed Feature Location Articles

Table A.2 classifies all of the feature location articles included in the survey in Chapter 2 within the taxonomy defined in Table A.1.

Table A.2: Articles classified within the taxonomy.

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Antoniol 2006 [5]	R	D	ET	M	Ranked	Java, C/C++	P	grep, SR, FCA	Mozilla, Firefox, Chimera, XFig, ICEBrowser, JHotDraw
Antoniol 2005 [4]	R	D	ET	M	Ranked	C/C++	P	grep, FCA	Mozilla
Bohnet 2008 [24]	T	D, S	ET, DG	M	Visual	C/C++	P	None	(N/A)
Bohnet 2008 [23]	T	D, S	ET, DG	M	Visual	C/C++	P	None	(N/A)
Bohnet 2007 [22]	T	D, S	ET, DG	M	Visual	C/C++	P	None	(N/A)
Bohnet 2007 [21]	T	D, S	ET, DG	M	Visual	C/C++	P	None	gcc
Bohnet 2006 [20]	T	D, S	ET, DG	M	Visual	C/C++	P	None	Firefox
Bohnet 2006 [19]	T	D, S	ET, DG	M	Visual	C/C++	P	None	LandXplorer Studio
Buckner 2005 [35]	T	S	DG	M	Visual	Java	NE	None	Art of Illusion
Chen 2001 [38]	R	H	Rep	St	Ranked	C/C++	P	None	KDE
Chen 2001 [40]	T	S	DG	M	Visual	C/C++	NE	None	Mosaic
Chen 2000 [39]	R	S	DG	M	Manual	C/C++	P	None	Mosaic
Cleary 2009 [45]	R	T	SC, Doc	M	Set	Java	B	Other	Eclipse
Cleary 2007 [44]	R	T	SC, Doc	M	Set	Java	B	Other	Eclipse
Čubranić 2005 [218]	R	H	Rep	A	Ranked	Java	B, NE	None	Eclipse

Table A.2: (continued).

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Čubranić 2004 [219]	R	H	Rep	A	Ranked	Java	B	None	Eclipse
Čubranić 2003 [217]	R	H	Rep	A	Ranked	Java	B	None	Eclipse
de Alwis 2008 [55]	T	S, D, H	DG, ET, Rep	C, M	Set	Java	P	None	ArgoUML
de Alwis 2007 [56]	CS	S, D, H	DG, ET, Rep	C, M	Set	Java	NE	JQuery, Feret, Suade	jEdit
Eaddy 2008 [62]	R	D, S, T	SC, ET, DG	C,M	Set	Java	B	LSI, SPR, DFT, SR	Rhino
Edwards 2009 [66]	R	D	ET	St	Set	C/C++	P	Other	Apache httpd
Edwards 2006 [65]	R	D	ET	M	Ranked	C/C++	P	None	Gunner, Joint STARS
Egyed 2007 [69]	T	D	ET	M	Visual	Java	P	None	ArgoUML, GanttProject, Video-on-Demand player
Eisenbarth 2003 [76]	R	D, S	ET, DG	M	Set	Java, C/C++	E	None	XFig, Mosaic, Chimera, Agilent
Eisenbarth 2001 [75]	R	D, S	ET, DG	M	Set	C/C++	P	None	XFig
Eisenbarth 2001 [74]	R	D, S	ET, DG	M	Set	C/C++	P	None	XFig
Eisenbarth 2001 [73]	R	D, S	ET, DG	M	Set	C/C++	P	None	Mosaic, Chimera

Table A.2: (continued).

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Eisenberg 2005 [77]	R	D	ET	M	Ranked	Java	B	SR	HTMLUnit, HTTPUnit, Axion
Gay 2009 [85]	R	T	SC	M	Ranked	Java	B	None	jEdit, Eclipse, Adempiere
Grant 2008 [90]	R	T	SC	M	Ranked	C/C++	P	None	cook
Griswold 2000 [96]	T	T	SC	St	Visual	Icon	P	None	wine2html
Hill 2009 [103]	R	T	SC	M	Ranked	Java	B	Other	Rhino, jajuk, jBidWatcher, javaHMO
Hill 2007 [102]	R	S, T	SC, DG	M	Set	Java	NE	Suade	GanttProject, jBidWatcher, Freemind
Ibrahim 2003 [105]	CS	D	ET	M	Set	C/C++	P	None	Generate Index
Janzen 2003 [107]	T	S	DG	C, M	Set	Java	P	None	Jin
Ko 2006 [117]	CS	—	—	—	—	Java	NE	None	Paint
Koschke 2005 [118]	R	D, S	ET, DG	St	Set	C/C++	P	FCA	sdcc, cc1
LaToza 2007 [123]	CS	—	—	—	—	Java	NE	None	jEdit
Liu 2008 [129]	R	D, T	SC, ET	M	Set	Java	B	SITIR	jEdit, Eclipse
Liu 2007 [130]	R	D, T	SC, ET	M	Ranked	Java	B	LSI, SPR, PROMESIR	jEdit, Eclipse
Lukoit 2000 [133]	T	D	ET	M	Visual	C/C++	P	None	Joint STARS, Mosaic

Table A.2: (continued).

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Marcus 2005 [141]	CS	S, T	SC, DG	M	Ranked	Java, C/C++	NE	grep, LSI, ASDG	Art of Illusion, Doxygen
Marcus 2004 [142]	R	T	SC	M	Ranked	C/C++	NE	ASDG, grep	Mosaic
Petrenko 2008 [157]	R	T	SC	M	Set	Java, C/C++	NE	None	Eclipse, Mozilla
Poshyvanyk 2007 [160]	R	D, T	SC, ET	M	Ranked	Java, C/C++	B	LSI, SPR	Eclipse, Mozilla
Poshyvanyk 2007 [165]	R	T	SC	M	Set	Java	B	LSI	Eclipse
Poshyvanyk 2006 [159]	R	D, T	SC, ET	M	Ranked	Java, C/C++	B	LSI, SPR	Eclipse, Mozilla
Poshyvanyk 2006 [166]	T	T	SC	C	Ranked	Java	P	Other	Violet, Art of Illusion
Poshyvanyk 2006 [162]	T	T	SC	St	Ranked	Java	P	None	(N/A)
Poshyvanyk 2005 [163]	T	T	SC	St	Ranked	C/C++	P	None	WinMerge, Doxygen
Ratiu 2007 [170]	R	S, T	SC, DG	C, M, St	Set	Java	P	None	Java standard library

Table A.2: (continued).

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Ratiu 2006 [169]	R	S, T	SC, DG	C, M, St	Set	Java	P	None	Java standard library
Revelle 2009 [173]	CS	D, S, T	SC, ET, DG	M	Ranked	Java	NE	LSI, SITIR, Other	jEdit, Eclipse
Revelle 2005 [172]	CS	O	SC	M	Set	Java, C/C++	P	None	sort, Minesweeper
Robillard 2008 [176]	R	S	DG	M	Ranked	Java	B, E, NE	None	jEdit, JHotDraw, Azureus, Violet, LOCC
Robillard 2008 [178]	R	H	Rep	M	Set	Java	NE	None	Ant, Azureus, Hibernate, JDT-Core, JDT-UI, Spring, Xerces
Robillard 2007 [182]	R	O	SC	M	Manual	Java	NE	None	AVID, Jex, Redback, jEdit
Robillard 2007 [183]	CS	O	SC	M	Set	Java	NE	None	GanttProject, jajuk, jBidWatcher, Freemind
Robillard 2005 [175]	R	S	DG	M	Ranked	Java	P	None	JHotDraw, Azureus
Robillard 2005 [184]	T	O	SC	M	Set	Java	P	None	JHotDraw
Robillard 2004 [177]	CS	—	—	—	—	Java	NE	None	jEdit
Robillard 2003 [180]	R	O	Other	M	Set	Java	NE	None	jEdit, JHotDraw

Table A.2: (continued).

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Robillard 2003 [181]	T	S	SC	M	Manual	Java	NE	None	(N/A)
Robillard 2002 [179]	R	S	SC	M	Manual	Java	NE	None	AVID, Jex, NSC
Rohatgi 2009 [187]	R	D, S	ET, DG	C	Ranked	Java	B	None	Checkstyle, Weka
Rohatgi 2008 [186]	R	D, S	ET, DG	C	Ranked	Java	B	None	Checkstyle
Rohatgi 2007 [185]	R	D, S	ET, DG	C	Ranked	Java	B	None	Weka
Safyallah 2006 [190]	R	D	ET	M	Set	C/C++	P	None	XFig
Shao 2009 [200]	R	S, T	SC, DG	M	Ranked	C/C++	P	LSI	iVistaDesktop
Saul 2007 [197]	R	S	DG	M	Ranked	C/C++	B, NE	Suade	Apache httpd
Shepherd 2007 [201]	R	T	SC	M	Visual	Java	B	Other	jBidWatcher, javaHMO, jajuk, iReport
Shepherd 2006 [204]	R	T	SC	M	Visual	Java	B	Other	JHotDraw
Simmons 2006 [207]	CS	D	ET	M	Set	C/C++	NE	Other	Apache httpd
Trifu 2009 [215]	R	S	DG	V	Set	Java	B	None	JHotDraw
Trifu 2008 [214]	R	S	DG	V	Set	Java	B	None	JHotDraw
Van Geet 2009 [221]	CS	D	ET	M	Set	COBOL	E	None	Belgian banking software
Walkinshaw 2007 [222]	R	S	DG	M	Visual	Java	NE	None	JHotDraw, NanoXML, Freemind

Table A.2: (continued).

	Article	Analysis	Info.	Gran.	Results	Langs.	Eval.	Compared	Systems Evaluated
Weigand-Warr 2008 [224]	T	S	DG	M	Ranked	Java	P	None	jEdit
Wilde 2003 [226]	CS	D, S	ET, DG	M	Set	FORTRAN	NE	ASDG, SR	CONVERT3
Wilde 2002 [230]	R	D	ET	M	Set	C/C++	P	None	FoodFight
Wilde 2001 [225]	CS	D, S	ET, DG	M	Set	FORTRAN	NE	ASDG, SR	CONVERT3
Wilde 1996 [227]	CS	D	ET	M	Set	C/C++	E, NE	None	Visitor control program, graph display system, test coverage monitor
Wilde 1995 [229]	R	D	ET	M	Set	C/C++	E	None	(N/A)
Wilde 1992 [228]	R	D	ET	M	Set	C/C++	E	None	(N/A)
Wong 1999 [234]	R	D	ET	St, V	Set	C/C++	E	None	SHARPE
Xie 2006 [235]	T	T	SC	M	Visual	C/C++	P	None	(N/A)
Zhao [244] 2006	R	S, T	SC, DG	M	Ranked	C/C++	E	None	DC, UnRTF
Zhao [243] 2004	R	S, T	SC, DG	M	Ranked	C/C++	E	None	DC

Appendix B

Exploratory Study Instructions

This appendix contains the instructions given to the participants of the exploratory study presented in Chapter 3. These instructions, along with the source code used in the study, can be found online at http://www.cs.wm.edu/semeru/data/icpc09-feature-location/case-study-instructions.html.

B.1 Overview

In this "experiment," you will investigate the relevance of methods to a particular feature from jEdit. You will be presented with several lists, each containing 10 methods, and asked to determine how relevant they are to the implementation of a feature.

B.1.1 System

jEdit, version 4.3 pre16, is a programmer's text editor written in Java.

B.1.2 Feature

jEdit has a global option for configuring the cursor/caret to be "thick," i.e. two pixels wide instead of one pixel so it is easier to see. This global option can be set by going to Utilities → Global Options → Text Area and checking the box for "thick" next to the caret options. Then click "OK" or "Apply."

B.1.3 Running jEdit

To run jEdit from the command line, type "java -jar jedit.jar" while in the build directory. On Windows, double clicking on the jedit.jar icon also works.

B.2 Detailed Instructions

1. Download and unzip the source code for jEdit 4.3pre16.

2. Optionally, but highly recommended, create a new Java project in Eclipse using the jEdit source code. File → New → Java Project. Give the project a name, select "Create project from existing source," and browse to the unzipped jEdit code you downloaded.

3. Download the lists of methods to inspect.

4. For each method in the lists, classify its relevance to the thick caret global option feature as either *Relevant*, *Somewhat Relevant*, or *Not Relevant*. Use the following guidelines:

   (a) Method names that are similar to the words in the feature's description are good indicators of possibly relevant code, but the method's source code should be inspected to ensure the method is actually relevant to the feature.

   (b) Determine if the method is relevant to the feature by asking "Would it be useful to know that this method is associated with the feature if I had to modify the implementation of the feature in the future?"

   (c) If most of the code in the method seems relevant to the feature, classify the method as *Relevant*. If some code within the method seems relevant but other code in the method is irrelevant to the feature, classify the method as *Somewhat Relevant*. If no code within the method seems relevant to the feature, classify it as *Not Relevant*.

   (d) If unable to classify the method by reviewing its code, explore the method's structural dependencies, i.e. what methods call it and are called by it. If the

method's dependencies seem relevant, than the method probably is as well. In Eclipse, to find references to or the declaration of a method, right click on the method's name and select References → Project or Declaration → Project.

(e) For any method you classify and are still hesitant about the classification you chose, please provide a brief one sentence explanation of your decision.

5. Once you have decided your classification for a method, place an "X" in the row for the method under the appropriate column, as in the example below.

Table B.1: An example of classifying methods.

	Relevant	*Somewhat Relevant*	*Not Relevant*
package1.class1.method1		X	
package1.class1.method2	X		
package1.class2.method1		X	
package2.class3.method4			X
package2.class3.method5	X		
package2.class4.method6			X
package3.class5.method7			X
package3.class5.method8			X
package3.class5.method9	X		
package4.class6.method10			X

6. Review your classifications one more time. Since the same method may appear in multiple lists, ensure it is classified consistently in every list.

7. Return your results as an e-mail attachment when you are done.

Appendix C

Feature Coupling Study Instructions

This appendix contains the instructions given to the participants of the developer study on feature coupling presented in Chapter 5. These instructions, as well as the files necessary to follow them, are also available online at http://www.cs.wm.edu/semeru/data/feature-coupling-study.

C.1 Instructions

Thank you for volunteering your time to participate in this study. The time required to complete this task is approximately 1-2 hours. Feel free to take breaks if you need to, but try to keep track of the actual amount of time spent working on this study. A computer running Windows or Linux is required; unfortunately a piece of software used in the study does not work on Macs. All information that you provide will be kept strictly confidential. In this study, you will be asked to examine the source code of several pairs of features (behaviors or functionalities of a software system) and determine if the features are coupled to one another. In other words, you are being asked if there is some relationship or dependency between the two features.

The Eclipse IDE (integrated development environment) will be used in this study. If you are unfamiliar with Eclipse, you may still participate in this study as it will primarily

be used to view source code files.

Follow the instructions below to begin.

1. Download the questionnaire and answer the first three questions about your programming experience. The questionnaire is available in PDF for participants who are at W&M and want to write their responses on a print-out. Otherwise, the questionnaire is available as an Excel spreadsheet or text file, and responses can be saved directly to the file.

2. Download the appropriate file for your operating system (windows-study-files.zip, linux32-study-files.zip, or linux64-study-files.zip) which contains a copy of Eclipse that has been pre-loaded with all the necessary source code, plug-ins, and data you will need.

3. Unzip study-files.zip to a convenient location which will be referred to as STUDY_HOME.

4. Change directories to STUDY_HOME/eclipse and start Eclipse.

5. As Eclipse is loading, you will be asked to select a workspace. Select STUDY_HOME/workspace and click "OK."

6. If once Eclipse starts, a window pops up with the title "Usage Data Upload," you may select "Turn UDC feature off" and click Finish.

7. Once Eclipse loads, you will see three projects listed on the left: dbViz, iBatis, and Rhino. You can ignore the fact that they have compilation errors. Right click on each individually and select "Refresh." Alternatively, you can select each project and press F5. By refreshing the projects, you ensure the files are in sync with the file system.

8. Go to Window → Show View → Other → FLAT^3. Select "Features" and click ok. FLAT^3 is a tool that among other things, manages the links between features and the source code that implements them.

9. In the view that was just opened, click the "View Menu" button in the FLAT^3 view and select Open Concern Domain → dbViz.

10. dbViz is a tool for visualizing database tables in a format similar to UML diagrams. Listed in the FLAT^3 view are some of dbViz's features and the methods associated with each. Clicking on a method's name takes you to that method's source code.

11. For the following pairs of features, examine the code associated with each feature and answer the question "Are the two features coupled?" Answer either *Strong No*, *Weak No*, *Weak Yes*, *Strong Yes*, or *Unknown*.

    For example, two features could be coupled if a change to the code of one feature could impact the behavior or performance of the other feature. Base your answer on what you observe in the code of the two features, and please give some rationale for your answer. You may answer the questions in any order, and if in the course of the study you wish to change one of your previous answers because you gained more knowledge of the system, you may.

    Record your answers in the appropriate place on the questionnaire.

Table C.1: The dbViz feature pairs.

Feature 1	**Feature 2**
Connect to Database The user enters the location of the database and authentication information to connect to the database.	*Exit dbViz* The user exits dbViz.
Auto Arrange ER Diagram The tables in the diagram are automatically arranged.	*Undo/Redo* Undo the last command or redo a command that has been undone.
Import from Schema from Database The user provides the location of the database and authentication information to connect to the database and dbViz loads the database's tables.	*Import Schema from SQL* The user provides the location of an sql file and dbViz loads the database's tables.

Table C.1: (continued).

Feature 1	Feature 2
Add Table to ER Diagram The user adds a database table to the diagram.	*Remove Table from ER Diagram* The user removes a database table from the diagram, and dbViz also removes any relationships to the table.
Save/Load ER Diagram The user saves the currently open diagram and then loads another.	*Load Saved ER Diagram* The user loads an existing diagram.

12. Click the "View Menu" button in the FLAT^3 view and select Open Concern Domain → Rhino.

13. Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users. Listed in the FLAT^3 view are some of Rhino's features and the methods associated with each. Clicking on a method's name takes you to that method's source code.

14. For the following pairs of features, examine the code associated with each feature and answer the question "Are the two features coupled?" Answer either *Strong No*, *Weak No*, *Weak Yes*, *Strong Yes*, or *Unknown*. Base your answer on what you observe in the code of the two features, and please give some rationale for your answer. You may answer the questions in any order, and if in the course of the study you wish to change one of your previous answers because you gained more knowledge of the system, you may. Record your answers on the same questionnaire as above.

Table C.2: The Rhino feature pairs.

Feature 1	Feature 2
Unary + operator The unary operator converts its operand to the Number type.	*The Addition operator (+)* The addition operator performs string concatenation or numeric addition.

Table C.2: (continued).

Feature 1	Feature 2
The Addition operator (+) The addition operator performs string concatenation or numeric addition.	*The Subtraction Operator (-)* The subtraction operator performs numeric subtraction.
Date.prototype.toString Return a string value representing the date in the current time zone in human-readable form.	*Date.prototype.valueOf* Return a number which is the time value.
Unicode Format-Control Characters The characters in category "Cf" in the Unicode Character Database.	*ToPrimitive* Convert a value to a non-object type.
parseInt Produce an integer value dictated by interpretation of the contents of a string.	*parseFloat* Produce a number value dictated by interpretation of the contents of a string.
SQRT2 The number values for the square root of 2.	*Date.prototype.getTimezoneOffset* The difference between local time and UTC time in minutes.

15. Click the "View Menu" button in the Features view and select Open Concern Domain → iBatis.

16. The iBatis Data Mapper framework makes it easier to use a database with Java and .NET applications. The iBatis project is heavily focused on the persistence layer frameworks known as SQL Maps and Data Access Objects (DAO). Listed in the FLAT^3 view are some of Rhino's features and the methods associated with each. Clicking on a method's name takes you to that method's source code.

17. For the following pairs of features, examine the code associated with each feature and answer the question "Are the two features coupled?" Answer either *Strong No*, *Weak No*, *Weak Yes*, *Strong Yes*, or *Unknown*. Base your answer on what you observe in the code of the two features, and please give some rationale for your answer. You may answer the questions in any order, and if in the course of the study you wish to change one of your previous answers

because you gained more knowledge of the system, you may. Record your answers on the same questionnaire as above.

Table C.3: The iBatis feature pairs.

Feature 1	**Feature 2**
Data Sources Data sources manage connections to databases.	*JTA* Java Transaction API (JTA) specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system.
JDBC The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases.	*JTA* Java Transaction API (JTA) specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system.
Query Execute a query on the database.	*Max Results* The maximum number of records to return.
Update Execute an update on the database.	*Autogenerated keys* Automatically generate primary key fields.
SELECT Execute a select on the database.	*SQL Script* Run an SQL script.

18. You have completed the study. Please return your completed questionnaire. Thank you for participating. You may remove the files you downloaded for this study from your computer.

Appendix D

List of Features

This appendix contains lists of the names of the features from the various systems that were located in this dissertation's evaluations.

D.1 jEdit Features

The features of jEdit used in the exploratory study presented in Chapter 3.

Patch #1608486, *Support for "Thick" Caret*

Patch #1818140, *Edit History Text*

Patch #1923613, *Reverse Regex Search*

Patch #1849215, *Bracket Matching Enhancements*

D.2 Eclipse Features

The features of Eclipse used in the exploratory study presented in Chapter 3.

Bug #5138, Double-click-drag to select multiple words is broken

Bug #31779, UnifiedTree should ensure file/folder exists

Bug #19819, Add support for Emacs-style incremental search

Bug #32712, Repeated error message when deleting and file is in use

The features of Eclipse used in the study presented in Chapter 4. To see the actual bug reports, go to https://bugs.eclipse.org/bugs/show_bug.cgi?id=xxxxx, where xxxxx is one of the id numbers listed.

54771	66182	66914	67821
64498	66216	66923	67873
64882	66234	66947	67913
64887	66297	67168	67930
64893	66357	67297	67944
65074	66651	67314	68013
65183	66819	67413	68117
65217	66835	67427	68146
65637	66858	67437	68157
65704	66880	67468	
65948	66888	67716	
66157	66898	67789	

D.3 dbViz Features

The features of dbViz used in the study in Chapter 5. The code associated with each of these features was determined manually by Eaddy et al. [61, 63]. The mappings can be found online at http://www.cs.columbia.edu/~eaddy/concerntagger/.

Add Table to ER Diagram	Load Saved ER Diagram
AutoArrange ER Diagram	Print ER Diagram
Connect To Database	Remove Table From ER
Exit dbViz	Save/Load ER Diagram
Import Schema From Database	Start dbViz
Import Schema From SQL File	Undo/Redo
Import Schema	

D.4 Rhino Features

All of the features listed below were used in the studies presented in Chapter 5. The features in boldface were used in the study in Chapter 4. The number before the feature's name indicates the section of the ECMAScript specification (third edition) that defines the feature. The specification can be found online at http://www.ecmascript.org/docs.php. The code associated with each of these features was determined manually by Eaddy et al. [61, 63]. The mappings can be found online at http://www.cs.columbia.edu/~eaddy/concerntagger/.

7.1 - Unicode Format-Control Characters
7.7 - Punctuators
7.2 - White Space
7.3 - Line Terminators
7.4 - Comments
7.5.1 - Reserved Words
7.5.2 - Keywords
8.6 - Object Type
7.5.3 - Future Reserved Words
7.6 - Identifiers
7.8.1 - Null Literals
7.8.2 - Boolean Literals
7.8.3 - Numeric Literals
7.8.4 - String Literals
7.8.5 - Regular Expression Literals
7.9.1 - Rules of Automatic Semicolon Insertion
8.1 - Undefined Type
8.2 - Null Type
8.3 - Boolean Type
8.4 - String Type
8.5 - Number Type
8.6.1 - Property Attributes
8.6.2 - Internal Properties and Methods
8.7.1 - GetValue
8.7.2 - PutValue
9.1 - ToPrimitive
9.2 - ToBoolean
9.3 - ToNumber
9.3.1 - ToNumber Applied to String Type
9.4 - ToInteger
9.5 - ToInt32
9.6 - ToUint32
9.7 - ToUint16
9.8 - ToString
9.8.1 - ToString Applied to Number Type
9.9 - ToObject
10 - Execution Contexts
10.1.3 - Variable Instantiation
10.1.4 - Scope Chain and Identifier Resolution
10.1.5 - Global Object
10.1.6 - Activation Object
10.1.7 - This

11.13.2 - Compound Assignment

11.14 - Comma Operator

12.1 - Block

12.2 - Variable Statement

12.3 - Empty Statement

12.4 - Expression Statement

12.5 - if Statement

12.6 - Iteration Statements

12.6.1 - do-while Statement

12.6.2 - while Statement

12.6.3 - for Statement

12.6.4 - for-in Statement

12.7 - continue Statement

12.8 - break Statement

12.9 - return Statement

12.10 - with Statement

12.11 - switch Statement

12.12 - Labelled Statements

12.13 - throw Statement

12.14 - try Statement

13 - Function Definition

13.2.1 - [[Call]]

13.2.2 - [[Construct]]

14 - Program

15 - Native ECMAScript Objects

15.1 - Global Object

15.2 - Object Objects

15.2.3 - Properties of Object Constructor

15.2.4 - Properties of Object Prototype Object

15.3 - Function Objects

15.3.3 - Properties of Function Constructor

15.3.4 - Properties of Function Prototype Object

15.3.5 - Properties of Function Instances

15.4 - Array Objects

15.4.3 - Properties of Array Constructor

15.4.4 - Properties of Array Prototype Object

15.4.5 - Properties of Array Instances

15.5 - String Objects

15.5.3 - Properties of String Constructor

15.5.4 - Properties of String Prototype Object

15.5.5 - Properties of String Instances

15.6 - Boolean Objects

15.6.3 - Properties of Boolean Constructor

15.6.4 - Properties of Boolean Prototype Object

15.7 - Number Objects

15.7.3 - Properties of Number Constructor

15.7.4 - Properties of Number Prototype Object

15.8 - Math Object

15.9 - Date Objects

15.9.4 - Properties of Date Constructor

15.9.5 - Properties of Date Prototype Object

15.10 - RegExp Objects

15.10.1 - Patterns

15.10.2 - Pattern Semantics

15.10.4 - RegExp Constructor

15.10.5 - Properties of RegExp Constructor

15.10.6 - Properties of RegExp Prototype Object

15.10.7 - Properties of RegExp Instances

15.11 - Error Objects

15.11.3 - Properties of Error Constructor

15.11.6 - Native Error Types Used in This Standard

16 - Errors

8.6.2.1 - [[Get]]

8.6.2.2 - [[Put]]

8.6.2.4 - [[HasProperty]]

8.6.2.5 - [[Delete]]

8.6.2.6 - [[DefaultValue]]

15.1.1.1 - NaN

15.1.1.2 - Infinity

15.1.1.3 - undefined

15.1.2.1 - eval

15.1.2.2 - parseInt

15.1.2.3 - parseFloat

15.1.2.4 - isNaN

15.1.2.5 - isFinite

15.1.3.1 - decodeURI

15.1.3.2 - decodeURIComponent

15.1.3.3 - encodeURI

15.1.3.4 - encodeURIComponent

15.1.4.1 - Object

15.1.4.2 - Function

15.1.4.3 - Array

15.1.4.4 - String

15.1.4.5 - Boolean

15.1.4.6 - Number

15.1.4.7 - Date

15.1.4.8 - RegExp

15.1.4.9 - Error

15.1.4.10 - EvalError

15.1.4.11 - RangeError

15.1.4.12 - ReferenceError

15.1.4.13 - SyntaxError

15.1.4.14 - TypeError

15.1.4.15 - URIError

15.1.5.1 - Math

15.2.1.1 - Object()

15.2.2.1 - new Object()

15.2.3.1 - prototype

15.2.4.1 - constructor

15.2.4.2 - toString

15.2.4.3 - toLocaleString

15.2.4.4 - valueOf

15.2.4.5 - hasOwnProperty

15.2.4.6 - isPrototypeOf

15.2.4.7 - propertyIsEnumerable

15.3.1.1 - Function()

15.3.2.1 - new Function()

15.3.3.1 - prototype

15.3.4.1 - constructor

15.3.4.2 - toString

15.3.4.3 - apply

15.3.4.4 - call

15.3.5.1 - length

15.3.5.2 - prototype

15.3.5.3 - [[HasInstance]]

15.4.1.1 - Array()

15.4.2.1 - new Array(...)

15.4.2.2 - new Array(len)

15.4.4.1 - constructor

15.4.4.2 - toString

15.4.4.3 - toLocaleString

15.4.4.4 - concat

15.4.4.5 - join

15.4.4.6 - pop

15.4.4.7 - push

15.4.4.8 - reverse

15.4.4.9 - shift

15.4.4.10 - slice

15.4.4.11 - sort

15.4.4.12 - splice

15.4.4.13 - unshift

15.4.5.1 - [[Put]]

15.4.5.2 - length

15.5.1.1 - String()

15.5.2.1 - new String()

15.5.3.2 - fromCharCode

15.5.4.1 - constructor

15.5.4.2 - toString

15.5.4.3 - valueOf

15.5.4.4 - charAt

15.5.4.5 - charCodeAt

15.5.4.6 - concat

15.5.4.7 - indexOf

15.5.4.8 - lastIndexOf

15.5.4.10 - match

15.5.4.11 - replace

15.5.4.12 - search

15.5.4.13 - slice

15.5.4.14 - split

15.5.4.15 - substring

15.5.4.16 - toLowerCase

15.5.4.18 - toUpperCase

15.5.5.1 - length

15.6.1.1 - Boolean()

15.6.2.1 - new Boolean()

15.6.4.1 - constructor

15.6.4.2 - toString

15.6.4.3 - valueOf

15.7.1.1 - Number()

15.7.2.1 - new Number()

15.7.3.2 - MAX_VALUE

15.7.3.3 - MIN_VALUE

15.7.3.4 - NaN

15.7.3.5 - NEGATIVE_INFINITY

15.7.3.6 - POSITIVE_INFINITY

15.7.4.1 - constructor

15.7.4.2 - toString

15.7.4.3 - toLocaleString

15.7.4.4 - valueOf

15.7.4.5 - toFixed

15.7.4.6 - toExponential

15.7.4.7 - toPrecision

15.8.1.1 - E

15.8.1.2 - LN10

15.8.1.3 - LN2

15.8.1.4 - LOG2E

15.8.1.5 - LOG10E

15.8.1.6 - PI

15.8.1.7 - SQRT1_2

15.8.1.8 - SQRT2

15.8.2.1 - abs

15.8.2.2 - acos

15.9.5.32 - setMinutes

15.9.5.33 - setUTCMinutes

15.9.5.34 - setHours

15.9.5.35 - setUTCHours

15.9.5.36 - setDate

15.9.5.37 - setUTCDate

15.9.5.38 - setMonth

15.9.5.39 - setUTCMonth

15.9.5.40 - setFullYear

15.9.5.41 - setUTCFullYear

15.9.5.42 - toUTCString

15.10.2.1 - Notation

15.10.2.2 - Pattern

15.10.2.3 - Disjunction

15.10.2.4 - Alternative

15.10.2.5 - Term

15.10.2.6 - Assertion

15.10.2.7 - Quantifier

15.10.2.8 - Atom

15.10.2.9 - AtomEscape

15.10.2.10 - CharacterEscape

15.10.2.11 - DecimalEscape

15.10.2.12 - CharacterClassEscape

15.10.2.13 - CharacterClass

15.10.2.14 - ClassRanges

15.10.2.15 - NonemptyClassRanges

15.10.2.16 - NonemptyClassRangesNoDash

15.10.2.17 - ClassAtom

15.10.2.18 - ClassAtomNoDash

15.10.2.19 - ClassEscape

15.10.3.1 - RegExp()

15.10.4.1 - new RegExp()

15.10.6.1 - constructor

15.10.6.2 - exec

15.10.6.3 - test

15.10.6.4 - toString

15.10.7.1 - source

15.10.7.2 - global

15.10.7.3 - ignoreCase

15.10.7.4 - multiline

15.10.7.5 - lastindex

15.11.1.1 - Error()

15.11.2.1 - new Error()

15.11.4.1 - constructor

15.11.4.2 - name

15.11.4.3 - message

15.11.4.4 - toString

15.11.6.1 - EvalError

15.11.6.2 - RangeError

15.11.6.3 - ReferenceError

15.11.6.4 - SyntaxError

15.11.6.5 - TypeError

15.11.6.6 - URIError

15.11.7.1 - NativeError Constructors Called as Functions

15.11.7.2 - NativeError()

15.11.7.4 - New NativeError()

15.11.7.5 - Properties of the NativeError Constructors

15.11.7.6 - NativeError.prototype

15.11.7.7 - Properties of the NativeError Prototype Objects

15.11.7.8 - NativeError.prototype.constructor

15.11.7.9 - NativeError.prototype.name

15.11.7.10 - NativeError.prototype.message

D.5 iBatis Features

The features of iBatis used in the study in Chapter 5. The code associated with each of these features was determined manually by Eaddy et al. [61, 63]. The mappings can be found online at http://www.cs.columbia.edu/~eaddy/concerntagger/.

Statements
DataSources
JDBC
DBCP
JNDI
SimpleDataSource
StatementTypes
Query
SELECT
Update
INSERT
UPDATE
DELETE
Auto-GeneratedKeys
StoredProcedures
Arbitrary
ComposingSQL
GlobalVariableSubstitution
DynamicFragments
Blocks
PropertyId
Conditionals
isPropertyAvailable
isNotPropertyAvailable
isNull
isNotNull
isEmpty
isNotEmpty
isParameterPresent
isNotParameterPresent
ComparePropertyId
CompareValue
isEqual
isNotEqual
isGreaterThan
isGreaterEqual
IsLessThan
isLessEqual
Iteration
Prepend
RemoveFirstPrepend
Open
Close
Conjunction
SQLFragments
ParameterTypes
PrimitiveTypes
boolean
byte
short
int
long
Collections
JavaBeans
Caching
CacheModel

CacheFlushing

FlushInterval

FlushTriggers

Mutability

Serializability

Transactions

AutomaticTransactions

Batches

Configuration

Config

SQLMap

GlobalProperties

Namespaces

Aliases

Resources

Classes

Files

Streams

Readers

Properties

URLs

JTA

LoggingImplementations

TransactionManagers

Timeout

UserTransactions

EXTERNAL

MaxRequests

MaxSessions

MaxTransactions

float

In-Memory

WEAK

SOFT

STRONG

LRU

FIFO

OSCACHE

java.sql.Connection

java.sql.PreparedStatement

Log4J

java.sql.ResultSet

java.sql.Statement

JakartaCommonsLogging

JDKLogging

JDBC.DriverProperty

JDBC.ConnectionURLProperty

JDBC.UsernameProperty

JDBC.PasswordProperty

JDBC.DefaultAutoCommitProperty

Pool.MaximumActiveConnectionsProperty

Pool.MaximumIdleConnectionsProperty

Pool.MaximumCheckoutTimeProperty

Pool.TimeToWaitProperty

Pool.PingQueryProperty

Pool.PingEnabledProperty

Pool.PingConnectionsOlderThanProperty

Pool.PingConnectionsNotUsedForProperty

Driver.*Property

Scripts

XML

Arrays

Lists

Maps

MappingInputParameterstoSQL

InlineParameters

InlineParameterSyntax

ParameterMapping

CustomTypeHandlers

NumericScale

MappingOutputParametersfromSQL

ResultMapping

ResultMapInheritence

ResultMapAggregation

NestedSelects

GroupBy

ColumnIdentifiers

NullValueSubstitution

CustomRowHandlers

ByteCodeGeneration

ClassCaching

RequestCaching

StatementCaching

XMLSyntax

LazyLoading

SQLMapActivity

NullLogger

SQLScripts

Profiling

Testing

ErrorHandling

SimpleFragments

ParameterClasses

ResultSetType

Input

Output

ResultSets

FetchSize

StatementExecution

Query1:1

Query1:N

MaxResults

IsolationLevel

Sessions

Connections

Bibliography

[1] F. Abreu, G. Pereira, and P. Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 13–22, Zurich, Switzerland, 2000.

[2] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[4] G. Antoniol and Y. Guéhéneuc. Feature identification: A novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 357–366, Budapest, Hungary, 2005.

[5] G. Antoniol and Y.G. Guéhéneuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006.

[6] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, pages 298–308, Vancouver, British Columbia, Canada, 2009.

[7] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[8] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[9] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[10] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[11] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5):89–94, 1994.

[12] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th IEEE/ACM International Conference on Software Engineering (ICSE'94)*, pages 482–498, 1994.

[13] D. Binkley, G. Gold, M. Harman, Z. Li, and K. Mahdavi. An empirical study of the relationship between the concepts expressed in source code and dependence. *Journal of Systems and Software*, 81:2287–2298, 2008.

[14] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 121–130, 2009.

[15] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[16] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, 12(25):1226–1242, 1976.

[17] B. W. Boehm, J.R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, Los Alamitos, CA, 1976.

[18] S. Bohner. Impact analysis in the software change process: A year 2000 perspective. In *Proceedings of the International Conference on Software Maintenance (ICSM '96)*, pages 42–51, Monterey, CA, 1996.

[19] J. Bohnet and J. Döellner. Analyzing feature implementation by visual exploration of architecturally-embedded call-graphs. In *Proceedings of the International Workshop on Dynamic Systems Analysis (WODA'06)*, pages 41–48, Shanghai, China, 2006.

[20] J. Bohnet and J. Döellner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the ACM Symposium on Software Visualization (SOFTVIS'06)*, pages 95–104, Brighton, United Kingdom, 2006. ACM.

[21] J. Bohnet and J. Döellner. Cga call graph analyzer - locating and understanding functionality within the gnu compiler collection's million lines of code. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07)*, pages 161–162, 2007.

[22] J. Bohnet and J. Döellner. Facilitating exploration of unfamiliar source code by providing 2 1/2 d visualizations of dynamic call graphs. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07)*, pages 63–66, 2007.

[23] J. Bohnet and J. Döellner. Analyzing dynamic call graphs enhanced with program state information for feature location and understanding. In *Proceedings of the International Conference on Software Engineering*, pages 915–916, Leipzig, Germany, 2008.

[24] J. Bohnet, S. Voigt, and J. Döellner. Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC'08)*, pages 268–271, 2008.

[25] L. C. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[26] L. C. Briand, P. Devanbu, and W. L. Melo. An investigation into coupling measures for c++. In *Proceedings of the International Conference on Software engineering (ICSE'97)*, pages 412–421, Boston, MA, 1997.

[27] L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurements. *IEEE Transactions on Software Engineering*, 22(1):68–85, 1996.

[28] L. C. Briand, J. Wüst, J. W. Daly, and V. D. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of System and Software*, 51(3):245–273, 2000.

[29] L. C. Briand, J. Wüst, and H. Louinis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99)*, pages 475–482, 1999.

[30] L.C. Briand, W. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7):706–720, 2002.

[31] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on World Wide Web*, pages 107–117, Brisbane, Australia, 1998.

[32] M. Bruntink, A. van Deursen, T. Tourwé, and R. van Engelem. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Proceedings of the International Conference on Software Maintenance*, pages 200–209, 2004.

[33] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31:804–818, 2005.

[34] J. Buchta, M. Petrenko, D. Poshyvanyk, and V. Rajlich. Teaching evolution of open source projects in software engineering courses. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'06)*, pages 136–144, 2006.

[35] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. Jripples: A tool for program comprehension during incremental change. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 149–152, 2005.

[36] L. Carver and W. G. Griswold. Sorting out concerns. In *Proceedings of the OOPSLA'99 Workshop on Multi-Dimensional Separation of Concerns*, 1999.

[37] M. Cataldo, A. Mockus, J. A. Roberts, and J.D Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–978, 2009.

[38] A. Chen, E. Chou, J. Wong, A.Y. Yao, Qing Zhang, Shao Zhang, and A. Michail. Cvssearch: searching through source code using cvs comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 364–373, 2001.

[39] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th IEEE International Workshop on Program Comprehension (IWPC'00)*, pages 241–249, Limerick, Ireland, 2000.

[40] K. Chen and V. Rajlich. Ripples: Tool for change in legacy software. In *Proceedings of the International Conference on Software Maintenance (ICSM'01)*, pages 230–239, Florence, Italy, 2001.

[41] S. Chidamber, D. Darcy, and C. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, 1998.

[42] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[43] N. Churcher and M. Shepperd. Towards a conceptual framework for object oriented software metrics. *SIGSOFT Softw. Eng. Notes*, 20(2):69–75, 1995.

[44] B. Cleary and C. Exton. Assisting concept location in software comprehension. In *Proceedings of the 19th Psychology of Programming Workshop*, pages 42–55, 2007.

[45] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2009.

[46] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Assoc., New York, second edition, 1988.

[47] P. Comon. Independent component analysis, a new concept? *Signal Process.*, 36(3):287–314, 1994. 195312.

[48] W.L. Conover. *Practical Nonparametric Statistics*. Wiley, third edition, 1998.

[49] R. Cooley, B. Mobasher, and J. Srivastava. Web mining: Information and pattern discovery on the world wide web. In *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97)*, pages 558–567, 1997.

[50] D. Coppit, R. Painter, and M. Revelle. Spotlight: A prototype tool for software plans. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pages 754–757, 2007.

[51] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–396, 1989.

[52] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 99(2):684–702, 2009.

[53] B. Cornelissen, A. Zaidman, A. van Deursen, and B. Rompaey. Trace visualization for program comprehension: a controlled experiment. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09)*, pages 100–109, Vancouver, BC, Canada, 2009.

[54] D. Darcy and C. Kemerer. Oo metrics in practice. *IEEE Software*, 22(6):17–19, 2005.

[55] B. de Alwis and G. C. Murphy. Answering conceptual queries with ferret. In *Proceedings of the Proceedings of the International Conference on Software Engineering (ICSE'08)*, pages 21–30, 2008.

[56] B. de Alwis, G. C. Murphy, and M. P. Robillard. A comparative study of three program exploration tools. In *Proceedings of the International Conference on Program Comprehension (ICPC'07)*, pages 103–112, 2007.

[57] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artefact management systems. *ACM Transactions on Software Engineering and Methodology*, 16(4):13, 2007.

[58] A. De Lucia, R. Oliveto, and L. Vorraro. Using structural and semantic metrics to improve class cohesion. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'08)*, pages 27–36, 2008.

[59] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[60] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 118–128, London, United Kingdom, 2007.

[61] M. Eaddy. *An Empirical Assessment of the Crosscutting Concern Problem.* PhD thesis, Columbia University, 2008.

[62] M. Eaddy, A. V. Aho, G. Antoniol, and Y.G. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC'08)*, pages 53–62, Amsterdam, The Netherlands, 2008.

[63] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[64] J. Eder, G. Kappel, and M. Schreft. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt, 1994.

[65] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, 2006.

[66] D. Edwards, N. Wilde, S. Simmons, and E. Golden. Instrumenting time-sensitive software for feature location. In *Proceedings of the International Conference on Program Comprehension (ICPC'09)*, pages 130–137, Vancouver, British Columbia, 2009.

[67] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116 – 132, 2003.

[68] A Egyed. Resolving uncertainties during trace analysis. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'04)*, pages 3–12, Newport Beach, CA, 2004.

[69] A. Egyed, G. Binder, and P. Grunbacher. Strada: A tool for scenario-based feature-to-code trace detection and analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pages 41–42, 2007.

[70] A. Egyed and P. Grunbacher. Identifying requirements conflicts and cooperation. *IEEE Software*, 21(6):50–58, 2004.

[71] A. Egyed and P. Grunbacher. Supporting software understanding with automated traceability. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):783–810, 2005.

[72] A. Egyed, M. Heindl, S. Biffl, and P. Grunbacher. Determining the cost-quality trade-off for automated software traceability. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pages 360–363, Long Beach, CA, 2005.

[73] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM01)*, pages 602–611, Florence, Italy, 2001.

[74] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'01)*, pages 176–179, 2001.

[75] T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the International Workshop on Program Comprehension (IWPC'01)*, pages 300–309, 2001.

[76] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[77] A. D. Eisenberg and K. De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pages 337–346, Budapest, Hungary, 2005.

[78] K. El-Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[79] M. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the Workshop on Dynamic Analysis (WODA'03)*, pages 24–27, 2003.

[80] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'03)*, pages 90–101, 2003.

[81] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[82] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 13–23, 2003.

[83] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[84] J. Gao, J.Y. Nie, G. Wu, and G. Cao. Dependence language model for information retrieval. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 170–177, Sheffield, United Kingdom, 2004.

[85] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 351–360, 2009.

[86] R. Geiger, B. Fluri, H. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Fundamental Approaches to Software Engineering (FASE'06)*, pages 411–425, 2006.

[87] M. M. Geipel and F. Schweitzer. Software change dynamics: Which dependencies do matter? empirical evidence from 35 java projects. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 269–272, 2009.

[88] O Giroux and M. P. Robillard. Detecting increases in feature coupling using regression tests. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 163–174, Portland, Oregon, USA, 2006.

[89] N. E. Gold and K.H. Bennett. Hypothesis-based concept assignment in software maintenance. *IEE Proceedings-Software*, 149(4):103–110, 2002.

[90] S. Grant, J. R. Cordy, and D. B. Skillicorn. Automated concept location using independent component analysis. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'08)*, pages 138–142, 2008.

[91] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Bern, 2007.

[92] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, 2005.

[93] O. Greevy, S. Ducasse, and T. Girba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):425–456, 2006.

[94] O. Greevy, T. Girba, and S. Ducasse. How developers develop features. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 265–274, 2007.

[95] W. G. Griswold. Coping with crosscutting software changes using information transparency. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, 2001.

[96] W.G. Griswold, Y. Kato, and J.J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Proceedings of the Workshop on Multi-Dimensional Separation of Concerns*, 2000.

[97] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[98] M. Harman, N. E. Gold, R.M. Hierons, and D.W. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 11–21, 2002.

[99] S.G Hart and L.E. Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. *Human Mental Workload*, 1:139–183, 1988.

[100] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[101] S. Henry and K. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[102] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the International Conference on Automated Software Engineering (ASE'07)*, pages 14–23, 2007.

[103] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*, pages 232–242. Vancouver, Canada, 2009.

[104] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, Monterrey, Mexico, 1995.

[105] S. Ibrahim, N.B. Idris, and A. Deraman. Case study: Reconnaissance techniques to support feature location using recon2. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 371–378. IEEE Computer Society, 2003. 956502 371.

[106] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering (TSE)*, 31(3):213– 225, 2005.

[107] D. Janzen and K. Volder. Navigating and querying code without getting lost. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 178–187, 2003.

[108] H. Jiang, T. Nguyen, I. X. Che, H. Jaygarl, and C. Chang. Incremental latent semantic indexing for effective, automatic traceability link evolution management. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 59–68, L'Aquila, Italy, 2008.

[109] Z. Jin and A. J. Offutt. Coupling-based integration testing. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)*, pages 10–17, Montreal, Quebec, Canada, 1996.

[110] S.H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, second edition, 2003.

[111] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvaskyla, Finland, 1997. Springer-Verlag.

[112] L. A. Klein. *Sensor and data fusion: A Tool for Information Assessment and Decision Making*. SPIE Publications, Bellingham, WA, 2004.

[113] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[114] A.J. Ko, H.H. Aung, and B.A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the International Conference on Software Engineering (ICSE'05)*, pages 126–135, 2005.

[115] A.J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th IEEE/ACM International Conference on Software Engineering (ICSE'07)*, pages 344–353, 2007.

[116] A.J. Ko and B.A. Meyers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*, pages 301–310, Leipzig, Germany, 2008. 1368130 301-310.

[117] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[118] R. Koschke and J. Quante. On dynamic feature location. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pages 86–95, Long Beach, CA, USA, 2005.

[119] J. Kothari, D. Bespalov, S. Mancoridis, and A. Shokoufandeh. On evaluating the efficiency of software feature development using algebraic manifolds. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'08)*, pages 7–16, 2008.

[120] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'06)*, pages 93–102, 2006.

[121] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. Reducing program comprehension effort in evolving software by recognizing feature implementation convergence. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 17–26, Banff, Canada, 2007.

[122] S. Kramer and H. Kaindl. Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Transactions on Software Engineering and Methodology*, 13(3):332–358, 2004.

[123] T. D. LaToza, David Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 361 – 370, 2007.

[124] Y. S. Lee, B. S. Liang, S. F. Wu, and F. J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of the International Conference on Software Quality*, pages 81–90, Maribor, Slovenia, 1995.

[125] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.

[126] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[127] Z. Li. *Identifying High-Level Dependence Structures Using Slice-Based Dependence Analysis*. PhD thesis, King's College London, University of London, 2009.

[128] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 59–68, Banff, Canada, 2007.

[129] D. Liu, M. Brockmeyer, and S. Xu. Tag (trace+ grep): a simple feature location approach. In *Proceedings of the International Workshop on Program Comprehension through Dynamic Analysis (PCODA'08)*, pages 17–21, Antwerp, Belgium, 2008.

[130] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the International Conference on Automated Software Engineering (ASE'07)*, pages 234–243, Atlanta, Georgia, 2007.

[131] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th IEEE/ACM International Conference on Software Engineering (ICSE'06)*, pages 112 – 121, Shanghai, China, 2006.

[132] S. Lukins, N. Kraft, and L. Etzkorn. Source code retrieval for bug location using latent dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 155–164, Antwerp, Belgium, 2008.

[133] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. Tracegraph: Immediate visual location of software features. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 33–39, Washington DC, 2000.

[134] A. Marcus. Semantic driven program analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 496–473, Chicago, IL, 2004.

[135] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis'03)*, pages 27–36, San Diego, CA, 2003.

[136] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the Automated Software Engineering (ASE'01)*, pages 107–114, San Diego, CA, 2001.

[137] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):811–836, 2005.

[138] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 133–142, Budapest, Hungary, 2005.

[139] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[140] A. Marcus and V. Rajlich. Panel summary: Identifications of concepts, features, and concerns in source code. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, page 718, Budapest, Hungary, 2005.

[141] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 33–42, 2005.

[142] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, Delft, The Netherlands, 2004.

[143] M. Marin, L. Moonen, and A. van Deursen. Documenting typical crosscutting concerns. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'07)*, pages 31–40, 2007.

[144] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, 2004.

[145] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1), 2007.

[146] R. Martin. Oo design quality metrics-an analysis of dependencies. In *Proceedings of the Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, pages 1–8, 1994.

[147] K. O. McGraw and S.P. Wong. Forming inferences about some intraclass correlation coefficients. *Psychological Methods*, 1(1):30–46, 1996.

[148] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'09)*, pages 41–48, Vancouver, Canada, 2009.

[149] A. Mehta and G.T. Heineman. Evolving legacy system features into fine-grained. In *Proceedings of the 24th International Conference on Software Engineering*, pages 417–427, 2002.

[150] T. Mende, R. Koschke, and F. Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):143–169, 2009.

[151] T. M. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–27, 2007.

[152] H. Olague, L. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, 33(6):402–419, 2007.

[153] Andrzej Olszak and Bo Nørregaard Jørgensen. Remodularizing java programs for comprehension of features. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD'09)*, pages 19–26, 2009.

[154] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M.J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE'04)*, pages 776–786, 2004.

[155] M. Petrenko, D. Poshyvanyk, V. Rajlich, and J. Buchta. Teaching software evolution in open source. *IEEE Computer*, 40(11):25–31, 2007.

[156] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC'09)*, pages 10–19, 2009.

[157] M. Petrenko, V. Rajlich, and R. Vanciu. Partial domain comprehension in software evolution and maintenance. In *Proceedings of the International Conference on Program Comprehension (ICPC'08)*, pages 13–22, 2008.

[158] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[159] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the International Conference on Program Comprehension (ICPC'06)*, pages 137–146, Athens, Greece, 2006.

[160] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[161] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 469–478, Philadelphia, PA, 2006.

[162] D. Poshyvanyk, A. Marcus, and Y. Dong. Jiriss - an eclipse plug-in for source code exploration. In *Proceedings of the International Conference on Program Comprehension (ICPC'06)*, pages 252–255, Athens, Greece, 2006.

[163] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeyev. Iriss - a source code exploration tool. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 69–72, Budapest, Hungary, 2005.

[164] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimothy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[165] D. Poshyvanyk and D. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the International Conference on Program Comprehension (ICPC'07)*, pages 37–48, Banff, Alberta, Canada, 2007.

[166] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with google. In *Proceedings of the International Conference on Software Maintenance (ICSM'06)*, pages 334 – 338, Philadelphia, PA, 2006.

[167] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Software*, pages 2–9, July-Aug. 2004.

[168] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the International Workshop on Program Comprehension (IWPC'02)*, pages 271–278. IEEE Computer Society Press, 2002.

[169] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don't). In *Proceedings of the Working Conference on Reverse Engineering (WCRE'06)*, pages 83–92, 2006.

[170] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *Proceedings of the International Conference on Program Comprehension (ICPC'07)*, pages 91–102, Banff, Canada, 2007.

[171] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04)*, pages 432–448, Vancouver, BC, Canada, 2004.

[172] M. Revelle, T. Broadbent, and D. Coppit. Understanding concerns in software: Insights gained from two case studies. In *Proceedings of the International Workshop on Program Comprehension (IWPC'05)*, pages 23–32, St. Louis, Missouri, 2005.

[173] M. Revelle and D. Poshyvanyk. An exploratory study on assessing feature location techniques. In *Proceedings of the International Conference on Program Comprehension (ICPC'09)*, pages 218–222, Vancouver, British Columbia, Canada, 2009.

[174] M. P. Robillard. *Representing Concerns in Source Code*. PhD thesis, University of British Columbia, 2003.

[175] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11 – 20, Lisbon, Portugal, 2005.

[176] M. P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4), 2008.

[177] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889– 903, 2004.

[178] M. P. Robillard and B. Dagenais. Retrieving task-related clusters from change history. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'05)*, pages 17–26, 2008.

[179] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the International Conference on Software Engineering (ICSE'02)*, pages 406–416, 2002.

[180] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the International Conference on Automated Software Engineering (ASE'03)*, pages 225–234, Linz, Austria, 2003.

[181] M. P. Robillard and G. C. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the International Conference on Software Engineering (ICSE'03)*, pages 822–823, 2003.

[182] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):1–38, 2007.

[183] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An empirical study of the concept assignment problem. Technical report, McGill University, June 2007.

[184] M. P. Robillard and F. Weigand-Warr. Concernmapper: Simple view-based separation of scattered concerns. In *Proceedings of the Eclipse Technology Exchange at OOPSLA (ETX'05)*, pages 65–69, 2005.

[185] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. Feature location based on impact analysis. In *Proceedings of the International Conference on Software Engineering and Applications (SEA'07)*, 2007.

[186] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. An approach for mapping features to code based on static and dynamic analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC'08)*, pages 236–241, 2008.

[187] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. An approach for solving the feature location problem by measuring the component modification impact. *IET Software*, 3(4):292–311, 2009.

[188] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Feature driven browsing. In *Proceedings of the 2007 International Conference on Dynamic Languages: in Conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 79–100, 2007.

[189] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN - SIGSOFT workshop on Program analysis for Software Tools and Engineering*, ACM Press, editor, pages 46 – 53, 2001.

[190] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proceedings of the International Conference on Program Comprehension (ICPC'06)*, pages 84–88, 2006.

[191] R.A. Sahner and K.S. Trivedi. Sharpe: Symbolic hierarchical automated reliability and performance evaluator, introduction and guide for users. Technical report, Duke University, Sept. 1986.

[192] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *Proceedings of the International Conference on Software Maintenance (ICSM'04)*, pages 72–81, Chicago, IL, 2004.

[193] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Towards employing use-cases and dynamic analysis to comprehend mozilla. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 639–642, Budapest, Hungary, 2005.

[194] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 71–80, 2006.

[195] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1986.

[196] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):620, 1975.

[197] M.Z. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 15–24, 2007.

[198] T. Savage, M. Revelle, and D. Poshyvanyk. Flat3: Feature location and textual tracing tool. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 2010.

[199] S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Higher Education, New York, 5th edition, 2001.

[200] P. Shao and R. K. Smith. Feature location by ir modules and call graph. In *Proceedings of the ACM Annual Southeast Regional Conference*, Clemson, South Carolina, 2009.

[201] D. Shepherd, Z. P. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the International Conference on Aspect Oriented Software Development (AOSD'07)*, pages 212–224, 2007.

[202] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proceedings of the 20th IEEE/ACM International Confernence on Automated Software Engineering (ASE'05)*, pages 193–202, 2005.

[203] D. Shepherd, L. Pollock, and E. Gibson. Design and evaluation of an automated aspect mining tool. In *Proceedings of the International Conference on Software Engineering Research and Practice*, 2004.

[204] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 3–14, Bonn, Germany, 2006.

[205] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Angineering*, pages 54–59, 2007.

[206] M. Sherriff and L. Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 268–277, 2008.

[207] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble. Industrial tools for the feature location problem: an exploratory study. *Journal of Software Maintenance: Research and Practice*, 18(6):457–474, 2006.

[208] R.M. Sirkin. *Statistics for the Social Sciences*. Sage Publications, CA, third edition, 2005.

[209] I. Sommerville. *Software Engineering*. Addison-Wesley, New Work, sixth edition, 2001.

[210] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[211] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[212] R. Tairas and J. Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, 2009.

[213] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[214] M. Trifu. Using dataflow information for concern identification in object-oriented software systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 193–202, 2008.

[215] M. Trifu. Improving the dataflow-based concern identification approach. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 109–118, 2009.

[216] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.

[217] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the International Conference on Software Engineering (ICSE'03)*, pages 408–418, Portland, OR, 2003.

[218] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[219] D. Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Learning from project history: a case study for software development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'04)*, pages 82–91, Chicago, Illinois, USA, 2004.

[220] A. Van Deursen and Hans-Gerhard Gross. An industrial case study in reconstructing requirements views. *Empirical Software Engineering*, 13(6):727–760, 2008.

[221] J. Van Geet and S. Demeyer. Feature location in cobol mainframe systems: an experience report. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 361–370, 2009.

[222] N. Walkinshaw, M. Roper, and M. Wood. Feature location and extraction using landmarks and barriers. In *Proceedings of the International Conference on Software Maintenance (ICSM'07)*, pages 54–63, Paris, France, 2007.

[223] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 461–470, Leipzig, Germany, 2008.

[224] F. Weigand-Warr and M. P. Robillard. Suade: Topology-based searches for software investigation. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*, pages 780–783, 2008.

[225] N. Wilde, M. Buckellew, H. Page, and V. Rajlich. A case study of feature location in unstructured legacy fortran code. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'01)*, pages 68–76, 2001.

[226] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.

[227] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proceedings of then International Conference on Software Maintenance (ICSM'96)*, page 270, 1996.

[228] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the International Conference on Software Maintenance (ICSM'92)*, pages 200–205, Orlando, FL, 1992.

[229] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[230] N. Wilde, S. Simmons, D. Edwards, and L. Pounds. But where does it do that? locating features in a distributed simulation. In *Proceedings of the Fall Simulation Interoperability Workshop*, Orlando, FL, 2002.

[231] F.G. Wilkie and B.A. Kitchenham. Coupling measures and change ripples in c++ application software. *The Journal of Systems and Software*, 52(2-3):157–164, 2000.

[232] K. Wong, S. R. Tilley, H. A. Mueller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.

[233] W. E. Wong and S. Gokhale. Static and dynamic distance metrics for feature-based code analysis. *Journal of Systems and Software*, 74(3):283–295, 2005.

[234] W.E. Wong, S.S. Gokhale, J.R. Horgan, and K.S. Trivedi. Locating program features using execution slices. In *Proceedings of the Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET'99)*, pages 194–203, 1999.

[235] X. Xie, D. Poshyvanyk, and A. Marcus. 3d visualization for concept location in source code. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*, pages 839–842, Shanghai, China, 2006.

[236] Y. Ye and G. Fischer. Reuse-conducive development environments. *Journal Automated Software Engineering*, 12(2):199–235, 2005.

[237] R. K. Yin. *Applications of Case Study Research*. Thousand Oaks. Sage Publications, CA, USA, 2 ed edition, 2003.

[238] Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 293–299, Toronto, Canada, 2001.

[239] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008.

[240] A. Zaidman, B. Du Bois, and S. Demeyer. How webmining and coupling metrics improve early program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 74–78, 2006.

[241] C. Zhai and J Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems*, 22(2):179–214, 2004.

[242] J. Zhao. Measuring coupling in aspect-oriented systems. In *Proceedings of the 10th IEEE International Software Metrics Symposium (METRICS'04)*, pages 1–9, Chicago, USA, 2004.

[243] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: towards a static non-interactive approach to feature location. In *Proceedings of the International Conference on Software Engineering (ICSE'04)*, pages 293–303, 2004.

[244] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodologies*, 15(2):195–226, 2006.

[245] Y. Zhou, H. Leung, and B. Xu. Examining the Potentially Confounding Effect of Class Size on the Associations between Object-Oriented Metrics and Change-Proneness. *IEEE Transactions on Software Engineering*, 35(5):607–623, 2009.

[246] T. Zimmerman and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*, pages 531–540, Leipzig, Germany, 2008.

[247] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[248] L. Zou, M. W. Godfrey, and A. E. Hassan. Detecting interaction coupling from task interaction histories. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 135–144, Banff, Alberta, Canada, 2007.

VITA

Meghan Revelle

Meghan Revelle is a PhD candidate at the College of William and Mary in Williamsburg, Virginia. She graduated *summa cum laude* with her B.S. in Computer Science from Mary Washington College in 2003, and she received her M.S. in Computer Science from William and Mary in 2005. Her research interests include software engineering, software maintenance and evolution, program comprehension, and reverse engineering.