Dynamic Feature Tracing: Finding Features in Unfamiliar Code

by

Andrew David Eisenberg
B. A., Rice University, 1998

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming to the required standard

The University of British Columbia
August 2004
© Andrew David Eisenberg, 2004
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Andrew Eisenberg
Name of Author (please print)

28/07/2004
Date (dd/mm/yyyy)

Dynamic Feature Tracing: Finding Features in Unfamiliar Code
Title of Thesis:

Master of Science
Degree:

Computer Science
Department of

The University of British Columbia
Vancouver, BC Canada
Year: 2004
Abstract

This dissertation shows that existing functional tests of software features can be used by a developer unfamiliar with the software system to identify source code relevant to those features as well as understand why the source code is relevant.

There have been prior techniques to locate features in source code which can be roughly broken down into techniques that use static analysis and those that use dynamic analysis.

Features, being behaviors of a system, are dynamic in nature. Therefore, this dissertation focuses on dynamic techniques, rather than the use static techniques.

The dynamic techniques all require significant knowledge about the system before the technique can be useful. Furthermore, they all suffer in one or both of these respects: they make binary judgments about which source code artifacts relate to a feature without determining to what extent they relate (meaning that the feature must be precisely characterized, or else the technique will return inaccurate results), or they do not provide an idea of why a piece of code is relevant to a feature.

Our technique of creating *Dynamic Feature Traces* improves upon previous work 1) by taking advantage of an existing test suite thereby reducing the amount of system knowledge necessary to use the technique, 2) by ranking the source code artifacts by how strongly they are related to the feature, and 3) by retaining some part of the execution trace of the test suite so that developers can understand why a piece of code is part of a feature. We show that our technique provides benefit over other techniques by applying it to existing software systems and comparing its results to an existing technique.
Contents

Abstract ii

Contents iii

List of Tables v

List of Figures vi

Acknowledgements vii

Dedication viii

1 Introduction 1
   1.1 The quagmire that is feature understanding ............. 2
      1.1.1 There is hope ................................ 2
   1.2 Previous attempts .................................. 3
   1.3 Enter: Dynamic Feature Traces ....................... 4
   1.4 Structure of the remainder of the dissertation ....... 5

2 Background 6
   2.1 Feature Location Techniques .......................... 6
      2.1.1 Static Feature Discovery ...................... 7
      2.1.2 Dynamic Feature Discovery .................... 9
   2.2 Test Suite Design Using Test Driven Development ..... 13
   2.3 Conclusions ...................................... 14

3 What is a DFT and how is it created? 16
   3.1 Partitioning the Test Suite ....................... 18
   3.2 Gathering the Execution Trace .................... 20
   3.3 Ranks and Calls ................................ 21
      3.3.1 Generating the Calls ....................... 21
List of Tables

2.1 Breakdown of Feature Location Techniques. Note that all previous techniques described here make binary judgments as to what code elements are relevant; no judges to what extent code elements are relevant. .......................................................... 15

4.1 Features Analyzed. RFE means "Request for Enhancement". ..... 35
4.2 Results of Quantitative Evaluation: This table shows the clustering of interesting methods in the top rankings. ...................... 38
4.3 Comparison of DFT results to a Software Reconnaissance approach. 41
4.4 Optimal Weightings of Heuristics for each Feature: The second, third, and fourth columns provide the optimal weightings of the three heuristics we implemented for the set of interesting methods provided in each of the features. The final column provides the ratio of the actual number of methods found to the expected number of methods found. .......................................................... 49
List of Figures

1.1 A DFT is created by structuring the footprint of a test set in terms of the footprint's calls and a ranking of its methods. .................. 4

3.1 The process for creating a Dynamic Feature Trace .................. 28
3.2 The results of a JQuery query that asks for all readers and writers of field \( x \). ......................................................... 29
3.3 A JQuery Package Browser with a Creators sub-query (showing all the creators of the Canvas class). The context menu has a list of additional queries that can be shown as sub-trees of the original query. 30
3.4 Part of the All Features query, showing the Animate feature. The Dynamic stack forward and Dyna stack back sub-queries show the dynamic call graph starting from the selected node. .................. 31

4.1 The static call graph emanating from performEvent. ................. 45
4.2 The dynamic call graph emanating from performEvent. Understanding how the two shaded methods were called helped us understand Javascript properties in general. ................................. 46
Acknowledgements

There are so many people who have helped, guided, prodded, or supported me in some way to finish. I am grateful to all of them.

Many thanks to Kris De Volder, my supervisor, for always listening to my ideas, no matter how silly or unformed they may have been. Gregor Kiczales, as second reader, has given me incredible advice on how to formulate my claims and tighten argument. Thanks to Gail Murphy for always promptly answering my questions. Also Jonathan Sillito needs to be thanked for providing me with the initial spark for the topic. Jan Hanneman, Rob Walker, and Kevin Wigger need to be thanked for their input on earlier drafts of this thesis.

In addition, my remaining lab mates must be thanked for their help. Chris Dutchyn and Brian De Alwis have always taken time out of their research to help me with \LaTeX and UNIX related questions. Doug Janzen and Reid Holmes (we made it!) need to be thanked for being with me to the end and sticking through the whole master's program with me. Mik Kersten must be thanked for always assuring me that it was not AspectJ's fault, but my own misunderstanding of the language.

I thank my family in New York for, although not really understanding what I do, realizing that it is important enough to me that I had to leave the city and country to come to Canada to continue my studies.

Lastly, although my cat, Bengali, who couldn't locate a feature even if it landed in his paw, has been there since the beginning and been kind to me even when things weren't going well.

ANDREW DAVID EISENBERG

The University of British Columbia
August 2004
To my wife and soulmate Stephenie, I could never have done this without you.
Chapter 1

Introduction

This dissertation shows that existing functional tests of software features can be used by a developer unfamiliar with the software system to identify source code relevant to those features as well as understand why the source code is relevant.

There have been prior techniques to locate features in source code which can be roughly broken down into techniques that use static analysis and those that use dynamic analysis.

Features, being behaviors of a system, are dynamic in nature. Therefore, this dissertation focuses on dynamic techniques, rather than the use static techniques.

The dynamic techniques all require significant knowledge about the system before the technique can be useful. Furthermore, they all suffer in one or both of these respects: they make binary judgments about which source code artifacts relate to a feature without determining to what extent they relate (meaning that the feature must be precisely characterized, or else the technique will have questionable usefulness; a problem discussed further in Section 4.2), or they do not provide an idea of why a piece of code is relevant to a feature.

Our technique of creating Dynamic Feature Traces improves upon previous work 1) by taking advantage of an existing test suite thereby reducing the amount of system knowledge necessary to use the technique, 2) by ranking the source code artifacts by how strongly they are related to the feature, and 3) by retaining some part of the execution trace of the test suite so that developers can understand why a piece of code is part of a feature. We show that our technique provides benefit over other techniques to features by applying it to existing software systems and comparing its results to an existing technique.
1.1 The quagmire that is feature understanding

According to De Lucia et al, software comprehension constitutes 50% to 90% of software maintenance time [24]. This comprehension process includes the identification of the relationship between source code and software features.1

Poor, missing, or out-dated documentation can render even well-designed source code nearly incomprehensible. Because maintaining documentation is typically a low priority for software developers, it is often not a reliable source of information regarding the structure of the system [13].

This is a particular problem for developers who are unfamiliar with a system. They may know that a feature exists through their interactions with the system; but without extra help, it is extremely difficult for them to locate source code relevant to the feature.

1.1.1 There is hope

However, the growing popularity of incorporating testing frameworks, such as JUnit [26], into the development life-cycle is encouraging a form of self-documentation that would be impossible without the test suites [3]. The existence and structure of a test suite implicitly relates to the structure of the system it is testing.

In particular, systems created according the Test Driven Development strategy [2, 3] will tend to have a clear mapping between test cases and features, with each feature being exhibited by one or more test cases. One of the basic tenets of Test Driven Development is that test cases should be written to test a piece of functionality before the functionality is implemented, thus ensuring that all functionality has a test attached to it. There are some tests (often called acceptance tests) that are written to explicitly test the correctness of a particular feature.

We call the set of test cases that all focus on the same feature that feature's test set. We define the footprint of a test set to be all the methods executed by the test set when run.2 Assuming a correlation between tests and features, test set footprints have the potential to help map features to source code and therefore help with software engineering tasks such as evolution.

---

1A feature is defined as any behavior of the system that is observable by the external world. When we discuss locating a feature, we mean determining the mapping between the feature (a behavior) and the source code that implements it (static structure of the program).

2Note that [9] and [10] attach a footprint directly to the feature (or requirement), but we do not. Not all source code executed by a test set is relevant to its related feature, and we want to make this distinction explicit.
1.2 Previous attempts

Using test sets to find implementations of features is not a new idea, see Section 2.1. We briefly introduce Software Reconnaissance [34] to motivate our discussion of strengths and weaknesses of previous attempts at feature location using dynamic analysis.\(^3\)

Software Reconnaissance [34] is a technique to help software maintainers locate key source code for a particular functionality. The technique depends on the availability of two test sets: one that exhibits the feature and one that does not. After executing the two test sets, the technique subtracts the footprint of the non-exhibiting test set from the exhibiting footprint to determine the relevant blocks of code.

Although Software Reconnaissance has been shown to be useful in large case studies [35], we claim that using it to locate features falls short because its results are a list of blocks of code where each block is deemed equally relevant to the feature, thus having the following disadvantages:

- Instead of presenting the full footprint of the test set (which would typically contain too many elements to be comprehensible to a developer if presented as a flat list), Software Reconnaissance retains only the elements that are deemed most relevant, thereby discarding potentially useful information.

- In order not to exacerbate the above problem, Software Reconnaissance depends on having high quality test sets that precisely characterize the feature. Therefore to provide quality results, the exhibiting and non-exhibiting test sets must be precisely correlated. However, obtaining a suitable non-exhibiting test set is particularly difficult because there is no clear definition of what non-exhibiting means. If the test sets are too unrelated, then the results set will be large and vague. If the test sets are too similar, then the results set will be small, or worse, empty.

- A flat list does not help a developer determine why its elements are relevant to a feature. There is no context for any of the elements being included in the list of results and hence no way to understand how a block of code was called from within the context of the feature.

As described in Section 2.1 all other dynamic feature location techniques suffer from one or more of the above short-comings.

\(^3\)The static techniques, while useful for identifying some concerns of interest in the source code, are less useful for identifying features. Since we define features as behaviors of a system, and they are only apparent dynamically.
1.3 Enter: Dynamic Feature Traces

We introduce Dynamic Feature Traces (DFTs) which help locate features in source code. A DFT for a feature is a combination of two sets of artifacts created from the execution of a test suite partitioned into test sets. One set is the rankings which contains all methods in the test set footprint ranked by confidence of each method’s importance for the feature. The other set is the calls which contains the set of all unique calls occurring between methods in the footprint.

Figure 1.1 describes how a DFT is created from a code-base with a test suite. First, the developer partitions the test suite into test sets. It is a non-exhaustive partitioning—test cases may be left out of the partitioning. Then the developer executes the test suite in conjunction with the DFT tool, which calculates the footprint of all defined test sets and splits them into the ranks and calls sets. This process is described in more detail in Section 3.2.

DFTs avoid the problems outlined in Section 1.2 in the following ways:

- A DFT ranks methods according to a measure of relevance established by heuristics, rather than making a binary decision about which methods to re-
tain. The heuristics estimate to what extent the method is relevant to the feature. Thus, a DFT retains all methods of a footprint but still highlights the most relevant ones.

- Because the non-exhibiting tests are used to make gradual adjustments to the ranking of methods rather than removing them outright, our approach does not require the availability of precisely correlated non-exhibiting test sets.

- A DFT retains information about the dynamically observed call-graph and makes this available to the developer. This is helpful to determine why methods are included in a footprint.

We have created a prototype Dynamic Feature Tracing tool which creates DFTs from a JUnit test suite of a Java\(^4\) system, given a mapping of test cases to test sets.

We have used the DFT tool to locate code relevant to features in existing, but unfamiliar, software systems, and perform a feature enhancement to one of the systems. We have also compared a DFT to the results of applying Software Reconnaissance to the same features of unfamiliar systems.

1.4 Structure of the remainder of the dissertation

In the following chapter, Chapter 2, we outline recent work in feature location and describing the strengths and weaknesses of existing techniques. Chapter 3 details what a dynamic feature trace is, how it is created, and how it can be used. We describe the evaluation, results, and analysis of DFTs and its prototype creation tool in Chapter 4. In Chapter 5 we present lessons learned, future research, and concluding remarks.

\(^{4}\)Java is a trademark of Sun Microsystems
This dissertation claims that using an existing test suite to aid with feature location can provide benefits over previous feature location techniques. The survey in this chapter first overviews previous feature location techniques, breaking them down into two broad categories: those that use only statically determined links to locate features, and those that use some dynamic analysis. We then provide an overview of Test Driven Development as it relates to test suite construction and self-documentation of a software system.

2.1 Feature Location Techniques

There are several important distinctions between the approaches to feature location in software systems. The broadest distinction is between those that employ static analysis only and those that use some dynamic analysis. We therefore split this section accordingly, with Section 2.1.1 introducing the static analysis techniques and Section 2.1.2 introducing the dynamic analysis techniques.

For all of these techniques, the located feature is some subset of the base code of the system. Whereas the static techniques rely on user interactions to determine this subset, the dynamic techniques rely on the execution of test cases or scenarios\(^1\) to choose the subset. Thus, at the minimum, all the dynamic feature location techniques rely on a set of tests or scenarios (i.e. test sets) that all exhibit the feature to be located.

Within this static/dynamic distinction, we can also distinguish techniques that enable a developer to help determine why a code element is related to a feature versus those that do not. What we mean is that some techniques allow a developer to

\(^1\)Scenarios are executions of a system (as test cases are), but are meant illustrate a particular interaction with a system, rather than check for correctness. For our purposes, these two terms are interchangeable.
understand how code elements of a feature are related to each other (typically, this is some sort of dependency or call graph). Although the static techniques are largely integrated with an IDE and hence static relationships between code elements of the feature can be retrieved, they cannot show how the code is related within the context of the feature (i.e. static techniques cannot always find which static relationships are exercised dynamically when the system exhibits a feature). Some of the dynamic techniques simply return a list of elements exercised when a feature is exhibited, but do not explain why they were executed. The other dynamic techniques do allow a developer to explore relationships between code elements helping to provide an intuition as to why elements are related to a feature.

Another distinction is between those (dynamic) techniques that return the entire test set footprint versus those that return only a subset of this footprint. The dynamic techniques rely on tracing the execution of a test set. Some dynamic feature location techniques return only the part of the footprint deemed most related to the feature in question. Often the entire footprint is too large for a developer to fully comprehend. By returning only the most important part of the footprint, these techniques focus the developer’s attention on only the code elements most relevant to the footprint (if the test sets are precisely correct). However, if the test set is not chosen precisely, vital parts of the footprint can be inadvertently removed. Other dynamic techniques avoid this problem by returning the entire footprint. The danger with this tactic is that the entire footprint can be too large to be comprehensible by the developer.

The remainder of this section introduces the feature location techniques in terms of the distinctions described above.

2.1.1 Static Feature Discovery

This sub-section explores some examples of techniques that use static analysis to locate features in source code. They all use relationships that are inherent in the source code and do not execute the system.

However, since we define features as behaviors of a system, it seems unnatural to us to search for features statically, without ever actually experiencing the features by executing the program.

This problem manifests itself in that, without executing the target system, it is difficult to determine exactly how the code will behave and what code gets executed when a feature is exhibited. An instance of this problem is the difficulty in determining the runtime type of an object using static analysis alone. While it is theoretically possible to perform static analysis and determine exactly how a system will behave under a given input set, this is technically difficult and none of the work
described here attempts to do this.

The following techniques all use some sort of static analysis to locate features in the source code.

**Feature Extraction and Analysis Tool**  The *Feature Extraction and Analysis Tool*, or *FEAT*, is a tool created by Robillard, [29], which enables a developer to locate modular concerns and encapsulate them into a *Concern Graph* [30], a formal way to represent source code concerns\(^2\) of a system. The process is primarily a manual one for the developer, but the tool facilitates code exploration and provides mechanisms for maintaining consistency between the code and the concern graphs as the code base evolves. All elements of a concern must be added explicitly through the FEAT tool. The tool is primarily useful for specifying concerns using an abstract representation of the source code, rather than the source code itself. A Concern Graph stores a subset of the dependency graph of the code and allows a developer to understand the relationships between the code elements in the graph.

A developer using FEAT will explore and learn about the system as the tool is being used and therefore little prior knowledge about the system is necessary to get use from the tool.

Since FEAT is constructed through a developer’s exploration of the static code base, relationships only available at runtime can be difficult to discover. For example, the exact execution path taken when a feature is exhibited can be difficult or impossible to determine statically.

**Dependency Graphs**  Chen and Rajlich [7] introduce a semi-automated technique for using dependency graphs to locate features in software systems. The developer browses a statically derived abstract system dependency graph (*ASDG*) to determine code belonging to a feature. An ASDG is similar to the combination of a static call graph and a data flow graph.

The structure of the ASDG is dependent on the type of static analysis used. In particular, the construction of the ASDG, based on an algorithm in [17] does not handle languages with dynamic dispatch (this is an instance of the problem of finding dynamic relationships statically). Although other static analyzers can be used to create the ASDG, there is nonetheless some difficult in creating succinct and correct ASDGs.

Chen and Rajlich’s approach has similar advantages and disadvantages to FEAT. A developer need not be intimately familiar with the target system before using their approach—the developer will learn about the system while exploring

\(^2\)A concern is informally defined as anything of interest.
it. Also, a developer can quickly understand the static relationships between code elements selected from the ASDG. However, dynamic relationships and how the system actually behaves when executed are difficult to determine.

The Concern Manipulation Environment  The Concern Manipulation Environment (CME) [15] is a framework to assist in the incremental adoption of Aspect-Oriented Software Development. As such, one of its capabilities is to help software developers in the identification of software concerns or features which can later be extracted from the code-base. This capability is currently supported in the CME plug-in for Eclipse [16]. The plug-in allows a developer to query the code-base for code relevant to a given concern. It also allows code to be explicitly added to a concern. Similar to the techniques described above, static relationships of elements related to a feature are retained and explorable. However, the CME does not support the use of dynamic analysis to determine which code artifacts are part of a concern and therefore has the same difficulties that FEAT has when determining runtime behavior statically.

Automatic Inference of Concerns  Robillard and Murphy present an algorithm for automatic inference of Concern Graphs in [31]. Rather than using only the static structure of the program to determine code related to a feature, as the previous techniques do, Robillard’s technique automatically infers concerns from the investigation activities of developers. The technique records the path through code taken during a developer’s exploration of the system. And it uses a set of heuristics to translate this exploration data into one or more concerns which can then be viewed and edited in FEAT. Although this technique gathers data from different sources than the techniques explained previously, the relationships explored and returned are always those that can be statically determined. This technique is more automated than the previous ones, but has the same problem in that it cannot easily determine dynamic relationships between model elements.

2.1.2 Dynamic Feature Discovery  

In addition to Software Reconnaissance [34], introduced in Section 1.1 and further discussed in Section 4.2, there are other techniques for using dynamic analysis to extract features from source code. To a greater or lesser extent, each of these rely on a system expert to create surgically precise test cases from which feature location data can be retrieved dynamically. Also, many of these techniques do not structure pieces of the located feature in any meaningful way and they do not show
relationships between the code elements. Hence, the results are presented out of context and can be difficult to understand.

**Retaining only some artifacts executed by test set**

These next two techniques retain only some artifacts executed by a test set. They remove the less relevant artifacts by comparison with a non-exhibiting test set. The exhibiting test set is the set of test cases whose execution collectively exhibits a feature, while the non-exhibiting test set is the set of test cases whose execution precisely does not exhibit a feature.

**Software Reconnaissance** As mentioned in Section 1.2, Software Reconnaissance [35, 34] locates source code related to a feature by subtracting the results of the non-exhibiting test set from the exhibiting test set. The remaining source code artifacts are those that uniquely implement a feature.

To locate a feature, exhibiting test sets must be constructed in such a way that the feature and (as much as possible) only the feature is exhibited. Non-exhibiting test sets must be constructed so that they precisely do not exhibit the feature. However, exactly how to do this is never clearly stated. If the non-exhibiting test set is improperly determined, there is a risk that significant portions of code will be inadvertently removed from the results.

The two test sets need to be constructed precisely, thus precluding the use of this technique by those unfamiliar with the target system. Also, the results of this technique are a flat list of code blocks that are uniquely part of the feature (assuming precise test sets). There is no way to determine why these code blocks are important to the feature since these techniques do not retain any of the calling context when the feature was exhibited. This means that there is no way to see the relationships between the code blocks as they pertain to the execution of the feature.

**Execution Slices** Wong, et al [36] describe a process similar to Software Reconnaissance in that they both rely on exhibiting and non-exhibiting test cases to find blocks of code related to features. They expand on Wilde's technique, realizing that the heuristics used to find code related to features must vary depending on the goal of exploration (for instance, the technique is slightly different when searching for code unique to a single feature, than when searching for code common to multiple features). They also provide further hints on how to find exhibiting and
non-exhibiting tests: \(^3\) "the excluding tests should be as similar as possible in terms of the execution slice to the invoking tests so that as much common code can be filtered out as possible."

Again, finding the non-exhibiting tests in an unfamiliar system can be difficult, and there is a penalty for poorly chosen non-exhibiting tests. As in Software Reconnaissance, the technique does not return all code blocks in a footprint (and hence can remove potentially important information if the test sets are not chosen precisely), and does not retain information on how the different blocks of code in the results pertain to each other (there is no notion of why the blocks are important to the feature).

Retaining all artifacts executed by test set

These next techniques retain all artifacts from a test set execution. Hence they avoid the problem of removing potentially important elements of an execution trace that is apparent in previous techniques. However, because of this, if the returned footprint is too large, the developer using the technique will be overloaded with information, impacting the usefulness of the results.

All these techniques are lacking the ability to emphasize those code elements that are somehow most relevant to the feature. A technique that does this without removing any of the elements of lesser importance would be able to focus the developer's attention on a small subset of the code relevant to the feature, without removing any other code that is potentially relevant.

The first technique, \(\chi\)Vue, does not retain any part of the calling context of the elements deemed to be related to the feature, and hence the results are an unstructured list of code locations with no reference to how they were called or why they are important.

The next two techniques, by Egyed and Eisenbarth, et al, allow developers to view relationships between code elements. Although this helps with the organization of elements deemed relevant to a feature, it may not be sufficient. Because code for a feature tends to be tangled with other code (due to the existence of crosscutting concerns), it is very likely that not all code executed by the test set is equally relevant to the feature being analyzed. These techniques do help elucidate why a code element is part of a feature, but they do not help understand to what extent.

\(\chi\)Vue \ Agraval et al in [37] introduce \(\chi\)Vue, which locates features in source code by recording which blocks of code are executed by a set of scenarios or test cases.

\(^3\)They use the terms "invoking test" and "excluding test" to mean exhibiting test and non-exhibiting test respectively.
Originally developed to work on the Y2K problem [12], xVue is a commercially viable tool. Its GUI is fully functional and completely integrated with the xSuds tool suite of which xVue is a part. It allows a developer to see the footprint of a test set at different levels of granularity. For example, the developer can view the blocks executed by a test set of a feature at a block level, method-level, file-level, or directory-level.

Their tool does help the developer understand how the executed blocks of code are distributed throughout the system, but it does not help to understand how they are related to each other, nor does it structure the results so that code more important to the feature is emphasized.

Locating Features in Source Code  Eisenbarth, Koschke, and Simon [11] combine both static and dynamic techniques to determine source code relevant to features. This technique applies concept analysis over a set of code elements exercised by a scenario or test case to determine relationships between the code and the features. Concept Analysis is a formal means to analyze binary relations, and in this work, it is used to determine a mapping between executed code elements and the scenarios. The results of the concept analysis is then compared to the static structure of the code from which the feature is extracted. The results are displayed as a sub-graph of the full dependency graph and can be viewed in the Rigi tool [28].

Eisenbarth's approach is semi-automated, and requires human input, including a domain expert to construct precise scenarios that exhibit the features to be analyzed.

This technique does retain a concept of why a code element is part of a feature since the sub-graph related to the feature can be displayed in the Rigi tool. However, only static relationships are displayed. The Rigi tool does not show which relationships between code elements were exercised dynamically and which relationships are only statically present, but inactive when the feature is exhibited.

Automated Requirements Traceability  Egyed proposes using scenarios (similar to test cases) to perform requirements traceability in [9, 10]. The technique uses scenarios to trace functional requirements (similar to features) to source code elements and other types of artifacts such as UML Sequence Diagrams [5]. Using existing profiling tools, the scenarios are traced and then related back to the requirement they represent. The executed code is then deemed relevant to the requirement.

The approach is iterative and progressively refines user-generated hypotheses regarding which source code elements are relevant to which functional requirements. With each iteration, the methods executed are retained, as well as the calls between
them. Since the approach is iterative, the user of the technique need not be intimately familiar with the system. Indeed, familiarity will increase with each iteration. However, a system expert must supply the initial scenarios.

This is the only technique which retains any part of the dynamic call structure pertaining to the feature being analyzed. It therefore retains information about relationships only visible during the execution of the system and allows a developer to browse them.

However, Egyed offers no evidence that the technique will scale to a large-scale system. Indeed, all the examples provided are Java systems containing less than 15 classes. We believe that a scalability problem arises because each element in the scenario footprint is deemed equally relevant to the feature, rather than organizing the code elements so that most relevant elements are emphasized.

2.2 Test Suite Design Using Test Driven Development

Test Driven Development is a development strategy where tests are written for new code before the code itself is written. Astels and Beck [2, 3] have each written a comprehensive book on the subject. We provide an overview of TDD in order to show that a test suite constructed using this strategy contains an implicit mapping between features and test cases. In Chapter 3 we show how we can leverage this implicit feature mapping to help developers locate features without them being familiar with the target system.

Under TDD, progress towards a larger goal ensues in small, incremental steps. One or more tests are written applying to an increment before any code is written. The new tests fail initially since the increment is not yet implemented. Work cannot proceed to the next increment until all tests pass. Thus, a test suite grows as the system itself grows and the test suite comprehensively tests all the increments of the system. If a failure is later found in the system, a failing test is written to expose it. This test is used to help find and fix the fault. And after the fault is fixed, the, now passing, test remains in the test suite, documenting the fix.

Developing software in this manner ensures that all aspects of a system: every feature, development increment, design decision, and bug fix are embodied in the test suite. Unfortunately, the more tests that are created, the more difficult it is to understand how they relate to the system they are testing and whether or not the tests are actually succeeding.

To overcome these difficulties, tools have evolved to help organize and explore the test suite of a system. Most allow tests to be organized in a hierarchical manner (often mirroring the object hierarchy of the code itself). They also provide a visual
representation of the success or failure of any tests.

The general family of testing frameworks is called XUnit, [8] with JUnit for Java, NUnit for .Net\(^4\), PyUnit for Python, and virtually any other programming language). They all provide a developer with a framework from which a general test suite can be written for a piece of software written in the target language of the XUnit tool.

Since these tools allow tests to be written in a hierarchical manner, test suites are often structured such that they mirror the structure of the system they are testing. Although such a hierarchy may not be sufficient to group test cases together based on any arbitrary relationship (due to the existence of crosscutting concerns [22]), it is better than no organization at all.

Test driven development defines two types of tests: unit tests and acceptance tests [8]. Unit tests are written by the developers as they write the code and test individual software components. Acceptance tests are written or designed by the customer, so that if the system passes all acceptance tests, the customer is satisfied and can accept the system as it is. An acceptance test corresponds to either a functional requirement (a feature) or a non-functional requirement (such as security or performance). These acceptance tests can be defined within the unit test hierarchy or in a separate hierarchy depending on their relationship to the tested code.

Thus, systems developed using TDD tend to have a clear mapping between functional behavior and test cases, and with tool support this mapping can be understood even by developers unfamiliar with the system.

2.3 Conclusions

As shown in Section 2.1, all previous dynamic feature location techniques rely on extensive knowledge about the target system before they can be readily applied. Also, no static feature location technique can easily identify which dynamic relationships between source code elements are realized when a feature is executed. Only one technique (see Section 2.1.2) maintains any part of the dynamic links between source code elements in the context of a feature execution. However, this technique has potential scalability issues because it does not emphasize the source code most relevant to the feature.

Table 2.1 shows a summary of the related work described in this chapter.

We show in Chapter 3 how we can use a test suite from any system developed using a Test Driven Development strategy to help with feature location. Using an

---

\(^4\) .Net is a registered trademark of the Microsoft Corporation.
<table>
<thead>
<tr>
<th>Technique</th>
<th>Static</th>
<th>Dynamic</th>
<th>Retain All</th>
<th>Retain Some</th>
<th>Binary Decision</th>
<th>Notion of why</th>
<th>Other Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>User locate feature through exploration</td>
</tr>
<tr>
<td>Dependency Graphs</td>
<td>Section 2.1.1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>Can browse static relationships</td>
<td>User explores dependency graph to locate feature</td>
</tr>
<tr>
<td>CME</td>
<td>Section 2.1.1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>Can browse static relationships</td>
<td>Query based or exploration based</td>
</tr>
<tr>
<td>Automatic-inference of concerns</td>
<td>Section 2.1.1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>Can browse static relationships</td>
<td>Infer concern (feature) automatically based on exploration</td>
</tr>
<tr>
<td>Software Reconnaissance</td>
<td>Section 2.1.2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>No</td>
<td>Difficulty in choosing non-exhibiting test set</td>
</tr>
<tr>
<td>Execution Slices</td>
<td>Section 2.1.2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>No</td>
<td>Difficulty in choosing non-exhibiting test set</td>
</tr>
<tr>
<td>xVue</td>
<td>Section 2.1.2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>No</td>
<td>Cannot relate results back to code structure</td>
</tr>
<tr>
<td>Locating using Concept Analysis</td>
<td>Section 2.1.2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Can browse static relationships</td>
<td>Relate results back to static structure only</td>
</tr>
<tr>
<td>Auto Requirements Traceability</td>
<td>Section 2.1.2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Can browse relationships determined dynamically</td>
<td>Potential scalability issues</td>
</tr>
</tbody>
</table>

Table 2.1: Breakdown of Feature Location Techniques. Note that all previous techniques described here make binary judgments as to what code elements are relevant; no judges to what extent code elements are relevant.

existing test suite helps to alleviate the reliance on knowledge of the target system before feature location techniques can be used.
Chapter 3

What is a DFT and how is it created?

In Chapter 2, we identified two methods of data collection for feature location techniques:

- Static analysis methods for feature location are tool supported, user-driven approaches to partition source code into features. Although, these approaches require significant time and effort on the part of the developer, they do not require significant knowledge about the target system prior to analysis. This is because the developer learns about the system as it is being explored.

- Dynamic analysis methods for feature location are test case or scenario driven. In these approaches, the data is collected from the execution trace of a test suite or a set of scenarios. Although these approaches tend to be more automated than the static analysis approaches, they tend to rely on precise input that can only be defined by someone with intimate knowledge of the target system in order to find code related to the particular feature. Thus, this defeats the purpose of using feature location techniques in the first place.

We also identified two design decisions for the dynamic analysis approaches to feature location:

**Amount of test set footprint retained** Some approaches retain only the most relevant parts of the test set footprint, while others retain all of it. The benefit of retaining only some is that the result set is smaller and potentially easier to understand, but the danger is that important code will be removed from the final results. Approaches that retain all of a footprint can potentially return too much data so that it would be overwhelming for the developer.
Structured or Unstructured results Some approaches simply return a portion (or all of) the test set footprint but do not provide any further structure. Other approaches provide the ability to understand the relationships between the elements of a test set footprint.

We believe that a strong feature location technique would combine many of the strengths of the above methods and design decisions. A more automated approach that does not require significant prior knowledge of the target system would combine the strengths of the dynamic and static data collection methods.

Also, it is possible to benefit from both sides of the trade-off between retaining all or some of the test set footprint. If the entire footprint is retained, but the most important elements are emphasized, the developer would not be overwhelmed by too much data, nor would potentially useful information be dropped. Thus the developer would have a basis to judge to what extent each element is important to a feature.

The ability of popular IDEs to provide many different visual representations of the relationships between code elements (e.g. call graphs and package browsers) is a benefit for developers. We believe that it is critical for the results of a feature location technique to be integrated with an IDE so that it can be viewed in the same way that a package browser or a call graph can. This would help a developer understand why an element is important to a feature.

The remainder of this chapter discusses Dynamic Feature Traces (DFTs) and how to create them, based on the principles outlined above. We elaborate on the part of the thesis statement that asserts that DFTs can be created with minimal prior knowledge of the system and then provide benefit for the developer by way of helping to map features to source code. This chapter provides a description of DFTs as artifacts as well as a description of how they are created. Since the creation of DFTs are intertwined with the artifact itself, we combine the description of the artifact with the description of its creation.

A Dynamic Feature Trace is comprised of two parts for each test set footprint related to a feature: 1) the ranks, an ordered set of all methods called during the execution of the test set footprint and 2) the calls, a set of all observed method calls amongst the methods in the test set footprint.

DFTs are created by the DFT tool in four steps, also shown in Figure 3.1:

Partitioning First, the developer determines which features to analyze and partitions the test suite accordingly, a process described in Section 3.1. This partitioning, represented in a user generated file is sent to the DFT parser.
Execution Tracing Then, the DFT analysis engine is applied to the target system. Using the partitioning created above, the analysis engine generates an execution trace of the test suite, a process described in Section 3.2.

Ranks and Calls Section 3.3 describes how the DFT analysis engine uses the execution trace to generate the ranks and calls. It details the heuristics used to rank the methods. This section also discusses feature finding heuristics that could be useful for inclusion in the tool, but have not yet been implemented.

Viewing of results Lastly, Section 3.4 describes how the rankings and calls sets can be integrated with existing IDEs and development processes. In particular, this section describes how DFTs can be viewed in JQuery [19] (a generic code browser, implemented as a plug-in for the Eclipse [18] IDE) to help visualize the code related to the features analyzed.

Therefor, the DFT tool consists of 3 parts: a parser to parse feature mapping files (Section 3.1), an analysis engine to generate the execution trace (Section 3.2) and generate the ranks and calls sets (Section 3.3), and an augmented version of JQuery that can import and apply DFT data (Section 3.4).

3.1 Partitioning the Test Suite

One of the goals of the DFT tool is to make the generation of test sets as simple as possible so that even a developer unfamiliar with the target system can use the technique. Therefore, rather than force a developer to create custom tests for the purpose of using the DFT tool, the tool expects a system with a large and comprehensive test suite that has a relatively clear mapping between features and test cases. This is not unreasonable for systems developed using a Test Driven Development strategy which often have these properties, as mentioned in Section 2.2. Even systems with incomplete test suites could still be analyzed by the DFT tool, but the results would be less accurate.

The developer must provide the DFT tool with a feature mapping (the exact structure of this mapping is described below), which groups test cases related to a feature into test sets. These are the features explicitly analyzed by the tool.

All remaining tests not explicitly mentioned in the mapping are implicitly grouped into test sets based on similarities in the test suite structure. Since the DFT tool is implemented as an extension of the JUnit [26] testing framework where similar test cases tend to be grouped into test classes, the DFT tool uses the test class as the implicit test set grouping. However, because this implicit test set grouping may not be perfect in all situations (e.g. test suites can be incomplete, poorly modularized,
or they focus on modules rather than features), the developer can explicitly exclude test cases or test classes from the analysis.

Using an implicit mapping of test cases to test sets means that there is no need to specify non-exhibiting test sets, since they are automatically generated.

The artifact created at this stage is a feature mapping file which defines a one-to-many mapping from features to test cases (i.e. each test case can only be part of one feature). It is created by the developer and user of the tool. In the file presented below, a mapping from feature to test classes is denoted by “C:” and a mapping from feature to test methods is denoted by “M:”. Classes and methods can be explicitly excluded from analysis by adding to the special EXCLUDE test set. If a method and its class are mentioned in two separate features, then the method overrides the class.

The following is an example feature tracing file for the same simple drawing application:

1. DrawCircular (  
4. )  
5.  
6. Animate (  
7.         C: ca.ubc.cs.draw.TestAnimate  
8. )  
9.  
10.EXCLUDE (  
12.         C: ca.ubc.cs.draw.TestDraw  
13.)

This feature tracing file identifies the DrawCircular (in line 1) and Animate (in line 6) features to trace, specifying two methods contributing to the former's exhibiting test set (lines 2 and 3) and one entire class contributing to the latter feature's exhibiting test set (line 7). It also specifies that all test methods in the test class TestDraw (line 12) not explicitly mentioned should be excluded from the final analysis. All other test classes are implicitly compared with the explicitly mentioned features using the heuristics described in Section 3.3. This mapping is parsed by the DFT parser and provided to the DFT analysis engine as input.
3.2 Gathering the Execution Trace

The DFT analysis engine is the part of the DFT tool that collects the execution trace of the system and generates the ranks and calls sets based on this execution trace. In this section we describe how the execution trace of the system is gathered.

The DFT analysis engine is a profiling and analysis tool written in AspectJ [21], an extension of the Java programming language that provides aspect-oriented programming facilities. As mentioned above, the DFT tool is an extension to the JUnit unit testing framework. During each unit test execution, the DFT analysis engine collects data on each method execution. Unless a unit test is EXCLUDED from the trace, all method calls and their call depth are stored by the analysis engine in separate partitions as defined in Section 3.1 above.

The DFT analysis engine is applied to a system by instantiating the main JUnit test suite of the system as a WeavableTestSuite, a class in the DFT library inherited from TestSuite which has advice that activates the profiling process. Hence, for most test suites, only one or two lines of code need to change for the DFT tool to be applied. The following code snippet creates a test suite for a simple drawing application and adds some test cases to the suite. The snippet is set up so that the DFT analysis engine will create a DFT upon the test suite's execution. Line 4 has been added and line 11 has been altered for this to occur.

```java
1. package ca.ubc.cs.draw;
2. import junit.framework.Test;
3. import junit.framework.TestSuite;
4. import ca.ubc.cs.testcaseaspects.WeavableTestSuite;
5.
6. public class AllTests extends TestCase {
7.   public static Test suite() {
8.     /* ADD REMAINING TEST CLASSES HERE */
9.     TestSuite suite = new WeavableTestSuite();
10.    suite.addTestSuite(TestDraw.class);
11.    suite.addTestSuite(TestMove.class);
12.    suite.addTestSuite(TestAnimate.class);
13.    return suite;
14. }
15. }
```

Typically, a JUnit test suite is created by instantiating an object of type
TestSuite. However, in order for the DFT analysis engine to be instantiated, the test suite must be of type WeavableTestSuite instead. The class WeavableTestSuite contains code that instantiates parts of the analysis engine. Also, the analysis engine contains aspects which define pointcuts that advise the WeavableTestSuite code. Line 4 has been added in order to import the WeavableTestSuite.

After altering the test suite to hook into the DFT analysis engine, the software system and test suite must be weaved and compiled using the AspectJ compiler, ajc.

The artifact created from this stage of the process is a data structure that contains all information needed to calculate the ranks and calls sets. This includes: a record of all method calls and the test case they are a part of, and the call depth of those calls. This information is then used by the analysis engine as described in the next section.

3.3 Ranks and Calls

Using the data gathered by the execution trace, we generate the ranks and calls sets.

3.3.1 Generating the Calls

The calls set is the simpler of the two sets to generate. For each test set defined in the mapping, one calls set is created. Based on the execution trace gathered by the analysis engine, call information is added to the calls set. Methods are added only once for each caller/callee pair in a particular test set, even if the same call occurs multiple times.

3.3.2 Generating the Ranks

The data required to generate the ranks set is also gathered from the execution trace. We rank the methods executed by each test set separately. Each method is given a score for each test set that it is a part of. The score is from 0 to 1—the average of three equally weighted heuristics. The final rank of a method is determined by sorting each exercised method by the sum of the individual scores. The top scoring method is given a rank of 1 and so forth. Ties are broken arbitrarily. The heuristics are as follows:

1Originally, we implemented the heuristics with equal weighting because we had no data to suggested otherwise and such a weighting was the most straightforward implementation. Section 4.4 describes the experiment we performed to determined a more optimal weighting of the heuristics.
**Multiplicity** A method exercised by a test set multiple times and in different situations is more likely to be important to the exhibited feature than a method used less often. We therefore chose to implement **Multiplicity**: the percentage of exhibiting tests that exercise a method compared to the percentage of methods in each non-exhibiting set of tests will contribute to its score.

Let \( curr \) = % of tests that exercises method in the current test set

\( \text{min} = \% \text{ of tests exercising method in the test set that exercises method fewest number of times (can be 0)} \)

\( \text{max} = \% \text{ of tests exercising method in the test set that exercises method greatest number of times} \)

\[
\text{Multiplicity} = \frac{curr - \text{min}}{\text{max} - \text{min}}
\]

**Specialization** A method that is exercised by many different test sets' exhibiting features is more likely to be a utility method and therefore not a part of any feature, than a method that is exercised by fewer test sets, which measures the **Specialization**.

\[
\text{Specialization} = (1 - \frac{\# \text{ of exercising test sets} - 1}{\# \text{ of number test sets}})
\]

Note that the \(-1\) is added to ensure that methods exercised by only a single test set has a score of 1.

**Call depth** We expect that for a reasonably well-designed and well-partitioned test suite, a test set will exhibit the behavior of the feature it focuses on in the most direct manner possible. All other test sets will tend to exhibit the feature less directly. We correlate directness with call depth.

We implemented this heuristic by comparing minimum call depths of an exercised method in all exercising test sets. The rationale is that the more directly a method is exercised, the shallower its call depth. To find the **Call Depth** score, \( cd \), for a method, \( m \), called with a stack depth of \( depth \) in a test set:
1. Determine the lowest and highest (low, hi) call depths (fewest or greatest number of stack frames from the top) for m in all test sets that exercise it.

2. Determine the linear scale, s, of depth between low and hi: 
   
   \[ s = 1 - \frac{\text{depth}_{\text{low}}}{\text{depth}_{\text{hi}}}, \text{ thus } s = 0 \text{ for the test set with } \text{low} \text{ call depth, and } s = 1 \text{ for the test set with } \text{hi} \text{ call depth. If } \text{low} = \text{hi} \text{ (i.e. all test sets have the same call depth), then } s = 500 \text{ (avoiding the division by 0).} \]

3. Next, re-scale s, taking into account all test sets that do not exercise m, using the Specialization score.

   \[ cd = (s \cdot (1 - \text{Specialization})) + \frac{\text{Specialization}}{\text{Specialization}} \]  

We assume that the non-exercising test sets of a method should have an effect on the Call Depth score. Accordingly, the fewer test sets that exercise a method, the higher the Call Depth score of the method in the test sets that do exercise the method. For this reason, the Call Depth score depends on the Specialization score.

Thus, the Call Depth score ranges from the Specialization score up to 1. The feature in which m has Call Depth = Specialization exercises m at the lowest call depth. While the feature with Call Depth = 1 exercises m at the highest call depth. In all other features, m has a Call Depth score linearly scaled between the two.

We do not assume that any single combination of the above heuristics will find all relevant methods in every situation. However, we do believe that each heuristic can find some relevant methods. We initially chose to weight all heuristics equally, but we explore other weightings in Section 4.4.

Other Heuristics

We identified other potential heuristics, but did not implement them for our studies. We include them here to illustrate that the heuristics we did implement were not arbitrarily chosen and also to motivate future work.

Naming Conventions Often methods related to a feature have names that are similar to that feature. For example, a method named draw() is very likely to be part of the implementation of a Draw feature. However, since naming conventions are often informal and the choice of a feature name would be independent of any naming convention (a feature name chosen by a developer
unfamiliar with the system would be unaware of any naming convention), it would be difficult to generalize this into a rule applicable for all features. We therefore chose not to implement this heuristic.

**Static Slicing** Using the definition of program slicing proposed by Weiser [33], starting from a test case that is part of a feature’s test set, we could remove some code that is irrelevant to the feature. Although this could very likely be a useful mechanism for removing some code, static slicing is very conservative in what it removes. However, what it does remove is code that could never be executed from a given starting point.

**Dynamic Slicing** A less conservative technique would be to use dynamic slicing [1, 23], which would likely return a smaller subset of the code when applied the execution trace of a test case that exhibits a particular feature. A dynamic slice as proposed by Agrawal [1] returns only statements executed given a particular input.

Although, not explicitly performed by the DFT tool, DFTs only contain methods that have been executed for a given set of inputs. For this reason, the generation of DFTs would not be improved by using a heuristic based on dynamic slicing.

**Inheritance Hierarchies** Types that are deeper in the inheritance hierarchy chain are by definition more specific and therefore more likely to be important to a feature than a type that is less deep. We could have implemented a heuristic that calculates the depth of a method in its inheritance hierarchy and determines a score based on that. Although, this heuristic could be a useful comparison between methods within the same hierarchy, it would be difficult or meaningless to compare methods of separate hierarchies. Therefore, we chose not to implement this heuristic.

**Multipliers** The intuition that interesting methods tend to follow other interesting methods led us to think of *multipliers*. A method called by or calling an interesting method (i.e. relevant to the feature) would itself be made more interesting. Although we did not implement this heuristic, we believe that it would be useful. However, the exact value of the multiplier would need to be experimentally determined. We do not yet have enough test data to explore this.

**Static Analysis of Incoming/Outgoing Calls** If we first statically determine all incoming calls to a method and then compare it to the incoming calls
actually exercised in a test set, we can use this information to determine the ranking. A method that is exercised by a large percentage of the total number of call sites is more likely to be important for a feature than a method that is not. A similar argument can be made for outgoing calls. This heuristic was not implemented, but it does seem promising.

**Profiling** Comparing the relative lengths of time spent in each method of a test set footprint could help infer a useful ranking. Perhaps, a method that takes a long time to execute would be more important for a feature than a quickly executing method. A heuristic using these measurements would also have to compare execution times of the same methods between features. Furthermore, the heuristic would need to account for threads that sleep in methods (due to, for example, I/O or concurrency issues).

Although none of the above heuristics have been implemented yet, they are topics for future research.

### 3.3.3 What does the DFT Tool Produce?

The artifacts of this stage of the DFT creation process are the two sets. Each element of the *ranks* set is a triple containing the feature name, the method name, and its rank in the feature. Each element of the *calls* set is another triple, containing the feature name, the caller method name, and the callee method name.

These files are implemented as *TyRuBa* [32] fact files. *TyRuBa* is the logic programming language—similar to Prolog—which serves as the underlying query language for *JQuery*, described next. JQuery provides a mechanism to view and interpret the two sets created by the DFT analysis engine.

### 3.4 Viewing Dynamic Feature Traces

The two sets that comprise the DFT are viewable in *JQuery* [20], a generic, extensible code browser built as a plug-in for the Eclipse IDE [18]. The two *TyRuBa* fact files (representing the *calls* and *ranks* sets) can be imported into JQuery where they add additional relationships to the existing source code artifacts.

---

2We do not detail the contents of the *TyRuBa* fact files because they are only an intermediate representation and understanding them will not help understand DFTs.
3.4.1 What is JQuery?

The core of JQuery is a logic database which contains a set of source code artifacts (fields, methods, classes, etc.) and some statically determined relationships between them (calls, reads, writes, inherits, etc.).

A developer can query the database. Each query asks for a set of Java source code artifacts that fulfill a boolean expression of relationships. A sample query, translated into plain English, is “Methods that write to or read from field X and the classes that contain these methods”. The result would be a set of 2-tuples with each 2-tuple containing a method (that writes to or reads from field X) and the class that contains it.

This result is displayed in a tree browser where each node represents one source code artifact and each edge represents one relationship. Figure 3.2 shows the results of a query that asks for all readers and writers of field _x (the x coordinate of a shape).

A developer can write general queries that contains the same structure as a package browser or an inheritance hierarchy browser. Or, queries can be specific and can return, for example, all the callers of a given method. Furthermore, a developer can incrementally expand the view by performing a sub-query on any node. These sub-queries perform a query using a result from a previous query as an input.

Default queries and sub-queries are included with JQuery and can be accessed from a context menu. These include a package browser, a call graph explorer, and an inheritance hierarchy browser as queries and calls to, calls from, overrides, overridden by as sub-queries. Figure 3.3 shows a query, sub-query, and the context menu, listing some of the default queries available in JQuery.

Double-clicking on any source code node opens an editor focused on that piece of code. A full description of the tool can be found in [19].

3.4.2 Extensions to JQuery

The two fact files contain new relationships that are added to the database. These relationships are the calls and the ranks of the methods in the DFT. Their creation has been described previously in this chapter. Since these relationships were determined using dynamic analysis, many of them could not easily have been determined using static analysis alone.

These new relationships can be queried in exactly the same way that the static relationships can, returning a browser containing code artifacts defined by relationships inherent in the DFT. A single query can combine both static relationships as well as DFT relationships. Thus, adding DFT data to a JQuery database
increases the semantic power of JQuery's queryable information.

We have extended JQuery so that DFT calls and ranks sets are properly inserted into the database, queries can be made on the DFT relationships amongst code artifacts, and queries that employ DFTs ranks are sorted by rank. We have also created some customized queries so that a developer can directly ask questions about a DFT:

1. **All Features**: this query returns a list of all the features and the methods exercised by these features. The resulting methods are in ranked order.

2. **Dynamic Stack Forward**: this sub-query uses the called methods facts to recreate a forward stack trace (with 5 frames maximum depth to prevent infinite loops) based on dynamic information retrieved from the DFT.

3. **Dynamic Stack Back**: similar to Dynamic Stack Forward except the stack is traced backwards a maximum of 5 frames.

Figure 3.4 shows an example of the queries that can be performed in JQuery with respect to DFTs. The top-level query is an All Features query, which returns the footprint of the test sets of all features analyzed by the DFT tool. In this case, the only feature visible is Animate. The animate() method is the top ranking method in the feature. From this method, a Dyna Stack Forward sub-query is shown, which displays a part of the execution trace starting from animate() within the Animate feature.

Because of the DFT tool's tight integration with JQuery and the Eclipse IDE, we believe this tool is more accessible and easier to use than a stand-alone version would be.

### 3.5 Summary

In this chapter, we have described the process of creating a DFT, what a DFT is composed of, and how DFTs can be integrated into an existing IDE. We have also shown how the process of creating DFTs attempts to overcome some of the problems we identified with previous feature location techniques. In the next chapter, we show to what extent DFTs achieve its goals by performing experiments the technique to evaluate the technique.
Figure 3.1: The process for creating a Dynamic Feature Trace
Figure 3.2: The results of a JQuery query that asks for all readers and writers of field \( x \).
Figure 3.3: A JQuery Package Browser with a Creators sub-query (showing all the creators of the Canvas class). The context menu has a list of additional queries that can be shown as sub-trees of the original query.
Figure 3.4: Part of the *All Features* query, showing the *Animate* feature. The *Dynamic stack forward* and *Dyna stack back* sub-queries show the dynamic call graph starting from the selected node.
Chapter 4

Evaluation

In Chapter 3, we introduced DFTs and explained how they are comprised of the \textit{calls} and \textit{ranks} sets which are created by using the DFT tool. We also explained the four step process of creating DFTs: partitioning (Section 3.1), execution tracing (Section 3.2), creating ranks and calls (Section 3.3), and viewing in JQuery (Section 3.4). This chapter focuses on determining to what extent DFTs convey relevant information about a feature and how they improve over existing mechanisms for feature location.

Of course, the claim that we would most like to make is that DFTs are unequivocally useful for developers. However, in order to substantiate such a claim, we would need to perform user studies on large software systems extending across development lifecycles. Unfortunately, we do not have the resources or time to execute this, so instead we make the following claims:

1. The heuristics used to rank methods in a DFT tend to cluster\textsuperscript{1} methods important to the feature in the upper ranks.

2. A developer unfamiliar with the target system can create a DFT that more completely captures the implementation of a feature (i.e. the source code) than the same developer (unfamiliar with the target system and using the same previously available test suite) could using other feature location techniques such as Software Reconnaissance. In fact, Software Reconnaissance breaks down in this situation: when there are no precise exhibiting and non-exhibiting test sets even though there is a relatively complete test suite available. However, applicable DFTs can still be created in this situation.

\textsuperscript{1}Informally, we mean that relevant methods are grouped in the top ranks. We define \textit{cluster} formally in Section 4.1.2.
3. DFTs capture knowledge about the implementation of features that can be applied towards expanding that feature; this information is not easily accessible from standard search techniques, such as those existing in the Eclipse IDE.

4. A justification for why a method is part of a feature can be retrieved from a DFT.

Claim 1 is substantiated by an experiment we performed to analyze the efficacy of the ranking heuristics, described in Section 4.1. DFTs were created by the author of this dissertation from three medium-sized open source systems that were developed using a Test Driven Development strategy (see Section 2.2). The DFTs were created by a developer unfamiliar with the target systems. We measured how well the DFTs predict methods necessary for a feature enhancement by comparing the rankings of the DFT to the actual methods changed based on data in a CVS [6] repository for the project. Thus, this experiment examined if DFTs effectively show to what extent a method is involved with the implementation of a feature.

We compared the Software Reconnaissance technique (as it appears in [35]) to DFT creation for the same set of feature enhancements described in Section 4.1, substantiating Claim 2. This is described in Section 4.2. This experiment examines how effectively the Software Reconnaissance technique can be applied to an unfamiliar system, in comparison to how DFTs can be created from the same unfamiliar system.

Claims 3 and 4 are discussed in Section 4.3, where the author of this dissertation took on the role of a developer and created DFTs from two of the open source systems used in the above experiments. The three case studies show how DFTs can be created and applied to particular feature enhancement tasks. We focus on how DFTs supply information not readily available using standard search techniques (substantiating Claim 3) and an examination of some of the dynamic call graphs of the features (substantiating Claim 4).

Finally, in Section 4.4, we examine alternate weightings for the three heuristics (Multiplicity, Specialization, and Call Depth). Throughout the previous experiments, we had weighted the heuristics equally, not because we believed that this would produce the strongest rankings, but because we had no evidence that any other weighting was more effective. In this section, we present an experiment where we re-weighted the heuristics to determine which heuristics contribute the most positively to the ranking.
4.1 Evaluation of Ranking Heuristics

In this section we describe an experiment that we performed to test the effectiveness of the DFT tool's ranking heuristics. We expect that a user of DFTs will tend to closely analyze the top results, but skim further down the list, ignoring the lowest results. Also, we do not expect all of the upper results to be interesting, but, rather, we do expect there to be a greater concentration (or clustering) of hits in the top-ranked elements. This is analogous to how a user might browse the results of a web search.

Therefore, we will measure effectiveness of the heuristics by analyzing how many methods deemed most relevant to the feature the tool places in the top \( x \) elements, for various \( x \). We call this measurement clustering. Relevance is defined in relationship to a particular feature enhancement task extracted from a CVS history as described below.

Methods can be relevant to a feature enhancement task in two ways. First, the core set of relevant methods are those that are directly modified (or deleted) in the process of implementing the enhancement. Second, there are methods which are directly linked to the core set because they call, are called by, override, or are in the same class as the core set. They are relevant as well because if suggested by the tool they will likely aid the discovery of the core set. We call the union of the core set with these directly linked methods the expanded set.

4.1.1 Data Collection

We chose three open source Java systems to analyze:

**HTMLUnit**\(^2\) [4] a unit testing framework focusing on testing dynamically generated web pages; about 17 KLOCs with 2,410 methods and constructors.

**HTTPUnit** [14] a unit testing framework that models the HTTP protocol; about 16.5 KLOCs with 2,227 methods and constructors.

**Axion** [25] a file-based database implemented in Java, 43 KLOCs, with 6,216 methods and constructors.

These three systems were chosen because their source repositories are freely available, they were developed using a Test Driven Development strategy (making their test-suites large and fairly comprehensive) and they all have an active feature request list from which we could relate feature enhancements to CVS check-ins.

From these systems, we searched for completed feature requests corresponding to changes in the source code. Table 4.1 provides a description of the features we
<table>
<thead>
<tr>
<th>System</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTMLUnit</td>
<td>RFE 805051</td>
<td>Added support for more operations on DOM nodes.</td>
</tr>
<tr>
<td>HTTPUnit</td>
<td>RFE 638311</td>
<td>Added support for document.open() and document.close() to Javascript.</td>
</tr>
<tr>
<td></td>
<td>RFE 717752</td>
<td>Added ability to retrieve page links by regular expression.</td>
</tr>
<tr>
<td>Axion</td>
<td>Issue 5</td>
<td>Fixed problem of boolean typed columns not being comparable.</td>
</tr>
<tr>
<td></td>
<td>Issue 9</td>
<td>Fixed problem of having multiple columns of type CLOB in a table.</td>
</tr>
</tbody>
</table>

Table 4.1: Features Analyzed. RFE means “Request for Enhancement”.

analyzed. The feature names we use are the same names as on the feature tracker hosted on the development sites. We use the feature names as they appear in the feature tracker to emphasize their being actual features of the systems. There was no prior knowledge of the code-base of HTTPUnit and Axion before the study was performed and there was minimal knowledge of the HTMLUnit code-base. Furthermore, there was little formal documentation provided for developers. All that existed were CVS logs, newsgroup discussions, bugzilla-like feature and bug trackers, and a test suite.

The process for analyzing the features was identical in all cases and performed by the author of this dissertation:

1. We read the feature request page for the feature we analyzed which includes information posted by a user about the desired addition or enhancement to the system.

2. We performed a checkout of the code from the CVS repository corresponding to the time immediately before the feature request was closed.

3. Using as much relevant information as possible, but without spending time to understand code details, we partitioned the test suite into an appropriate test set for the feature.
4. We compiled and wove the code with the DFT library, executed the test suite, and collected data based on the results of the execution.

5. The DFT analysis engine applied the ranking heuristics to the data collected during the execution (performed automatically upon completion of the test suite execution).

6. Using CVS tools, we generated the core and expanded sets of methods corresponding to the feature change.

7. We compared the DFT to the core and expanded sets.

We performed these experiments on a Pentium 4 with 764 Mb RAM and a 2.4 GHz processor running Windows XP. The AspectJ compiler used for this experiment was version 1.1.1. No more than 15 minutes were spent partitioning the test suite for each feature enhancement.

The actual execution time of the test suites increased by about twofold when the DFT analysis engine was woven into the code. HTTPUnit and HTMLUnit’s test suites each executed in over 2 minutes without the DFT analysis engine’s involvement, and just under 4 minutes each with the involvement. Axion’s test suite ran in just under one and a half hour without the analysis engine and about 3 hours with the analysis engine. This extremely long time is likely because of its heavy use of reading from and writing to the disk. The extra time needed to run the test suites was most likely due to the tracing mechanisms added to the methods in the system.

4.1.2 Analysis

Table 4.2 shows the results of our study. Each feature analyzed has three sets of numbers:

- **Num Found** The number of relevant methods found in the top 10, 20, 30, 40, and 50 elements respectively. Non-integers exist because we proportionally compute the ranks of equally scoring methods that cross the 10, 20, etc. boundaries.

- **Clustering** The raw count (from Num Found column) divided by the expected number of relevant methods that would be found by an even distribution.

The formula for calculating the expected number of methods $E[X]$ in the top $X$ ranked methods, for $T$ total methods and $I$ interesting methods is: $E[X] = \frac{I}{T^X}$. Therefore, $clustering = \frac{\text{NumFound}}{E[X]}$. 

36
We found that the raw count alone was hard to translate into a sense of whether a ranking should be considered effective or not. This is because it does not immediately convey a sense of the relative degree of clustering with respect to the size of the footprint and the actual number of interesting methods. A normalized number significantly greater than 1 indicates that the heuristics are effective—resulting in a high degree of clustering—whereas a number close to 1 signifies the heuristics are non-effective.

**Probability** The probability that a random distribution would produce a strictly better clustering than what the heuristics produce. These values were computed using a variant of the Hypergeometric Cumulative Distribution Function [27]. This number provides another means to gauge effectiveness by quantifying how difficult it is to do better.

As shown in Table 4.2, of the five feature enhancements in this study, the heuristics appear effective in the core set of three of them (RFE 805051, RFE 717752, and Issue 5). In each of these three, a large portion of the relevant core methods were clustered with high ranking. However, the results were less convincing for the expanded set of RFE 805051.

For RFE 638311, the heuristics performed poorly on the core set, but were moderately successful in picking out some of the relevant methods of the expanded set. In the core set, they failed to identify the single relevant method covered by the test cases.

There is one anomalous case (Issue 9) where the heuristics appear to be counter-productive. Trying to explain this anomaly, we discovered that we had missed a test case in the test set for the feature enhancement. This was not an initially obvious test case and had no apparent relationship to CLOBs (the data-type related to the feature). After adding the missing test case to the test set and re-running, the new ranking for Issue 9 Expanded was improved, but was still unable to rank any of the 4 core methods highly. We chose to not update the results presented in the table accordingly because the initial results more realistically reflect how the tool performs when wielded by a user with little prior knowledge of the target system.

Recall that the goal of this experiment was to establish a measure of effectiveness of the ranking heuristics used to create the DFT and not to assess the completeness of the coverage of relevant methods in the DFT. Indeed, this coverage cannot possibly be affected by changing the heuristics, since a DFT retains all methods in a test set footprint and the heuristics only alter their emphasis.

The number of relevant methods discovered is related to the quality of the test suite as well as the effectiveness of the partitioning. A deficiency in either can
### Table 4.2: Results of Quantitative Evaluation

This table shows the clustering of interesting methods in the top rankings.

<table>
<thead>
<tr>
<th>Feature</th>
<th># methods exercised</th>
<th>total # relevant methods</th>
<th>relevant methods found</th>
<th>$X = 10$</th>
<th>$X = 20$</th>
<th>$X = 30$</th>
<th>$X = 40$</th>
<th>$X = 50$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFE 805051</td>
<td>215</td>
<td>20</td>
<td>7</td>
<td>Num Found</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>6.14</td>
<td>3.07</td>
<td>2.05</td>
<td>3.46</td>
</tr>
<tr>
<td>Expanded</td>
<td>215</td>
<td>84</td>
<td>62</td>
<td>Num Found</td>
<td>3.5</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>1.21</td>
<td>2.08</td>
<td>1.62</td>
<td>1.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Probability</td>
<td>0.20</td>
<td>0.0005</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>RFE 638311</td>
<td>304</td>
<td>5</td>
<td>1</td>
<td>Num Found</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Expanded</td>
<td>304</td>
<td>61</td>
<td>40</td>
<td>Num Found</td>
<td>1.5</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>1.14</td>
<td>1.14</td>
<td>1.01</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Probability</td>
<td>0.23</td>
<td>0.26</td>
<td>0.36</td>
<td>0.13</td>
</tr>
<tr>
<td>RFE 717752</td>
<td>291</td>
<td>6</td>
<td>1</td>
<td>Num Found</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>0.00</td>
<td>7.27</td>
<td>9.7</td>
<td>7.27</td>
</tr>
<tr>
<td>Expanded</td>
<td>291</td>
<td>32</td>
<td>17</td>
<td>Num Found</td>
<td>0</td>
<td>0.5</td>
<td>3.5</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>0</td>
<td>0.42</td>
<td>2.00</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Probability</td>
<td>0.45</td>
<td>0.51</td>
<td>0.04</td>
<td>0.035</td>
</tr>
<tr>
<td>Issue 5</td>
<td>231</td>
<td>13</td>
<td>13</td>
<td>Num Found</td>
<td>0.33</td>
<td>1.67</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>0.59</td>
<td>1.48</td>
<td>2.96</td>
<td>3.11</td>
</tr>
<tr>
<td>Expanded</td>
<td>231</td>
<td>65</td>
<td>62</td>
<td>Num Found</td>
<td>3.33</td>
<td>6.66</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>1.24</td>
<td>1.24</td>
<td>1.74</td>
<td>1.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Probability</td>
<td>0.20</td>
<td>0.17</td>
<td>0.003</td>
<td>0.001</td>
</tr>
<tr>
<td>Issue 9</td>
<td>463</td>
<td>4</td>
<td>4</td>
<td>Num Found</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Expanded</td>
<td>463</td>
<td>73</td>
<td>64</td>
<td>Num Found</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clustering</td>
<td>0</td>
<td>0.18</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Probability</td>
<td>0.77</td>
<td>0.89</td>
<td>0.93</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Negatively impact the clustering of useful methods in the results. Still, even with a relatively naive partitioning, the heuristics appear to be effective.

Even though it is not the focus of this experiment, there are some observations worth mentioning about the discrepancies between the number of relevant methods of a feature (third column) versus the number of relevant methods actually
found (fourth column):

**RFE 805051** Of the 12 methods not found, 3 items not found were getters or setters, 4 were factory methods, and 3 were constructors. None of these 10 methods had code that perform anything more than get, set, create, or initialize. The remaining 2 methods had significant code in them, but two were located physically close to other methods that were found.

**RFE 638311** Of the 4 methods not found, none were executed from any test in the entire test suite, implying that the test suite was incomplete.

**RFE 717752** Of the 5 methods not found, all 5 of them were exercised in the JUnit test file HtmlTablesTest. Had this test class been included in the study, then all missing methods would have been found. When the heuristics are run again with the more accurate feature mapping, the clustering of relevant methods found becomes 14.6, 7.3, 8.9, 3.6, 3 for the Core set and 1.5, 1.9, 2.0, 1.9, 2.0 for the Expanded set. All relevant methods are exercised in the test set and it is a significant improvement.

Despite the heuristics not ranking highly all relevant methods for all features that we analyzed, we were able to create DFTs that clustered some of the methods in the top ranks despite being unfamiliar with the target systems.

Although there is variation amongst the results, the majority—4 out of 5 feature enhancements analyzed—suggest that the ranking heuristics in DFTs tend to cluster methods relevant to a feature in the upper ranks. Because these studies were performed with little prior knowledge of the target system, the tests chosen were not always the ones that most strongly exercised the feature. We did show that more familiarity with the system would produce a more precise partitioning and hence a stronger clustering, but even without familiarity, a reasonable partitioning can be created.

### 4.2 Comparison to Software Reconnaissance

We compared the DFTs to Software Reconnaissance as described in [34] in order to substantiate Claim 2. How many relevant methods can Software Reconnaissance discover given the same feature in the unfamiliar systems described in the previous section?

As described in Section 2.1.2, Software Reconnaissance is a technique to locate features in a software system. To use the technique, an exhibiting test set and a non-exhibiting test set for that feature are executed. Typically, these two test
sets are chosen by a system expert who has intimate knowledge of the system. The set of blocks executed\(^3\) by the non-exhibiting test set is subtracted from the set of blocks executed by the exhibiting test set. The remaining blocks of code are uniquely part of the feature. In this study, however, we apply Software Reconnaissance at the method level. We chose this because we wanted to be consistent with the DFT tool which also works at method granularity (we discuss potential ramifications of this choice later in the section).

For the same features examined in Section 4.1, the author of this dissertation created the exhibiting test set by using the same feature test set used to create the DFT and labelled all others as non-exhibiting (except for a small number of excluded test cases that were quickly deemed neither exhibiting nor non-exhibiting). Because there was little prior knowledge about the target systems, it would have been impractical to create a more precise partitioning into the exhibiting nor non-exhibiting test sets without the significant overhead of understanding the systems (thus defeating the purpose of using feature location techniques in the first place). Although this is not the standard way of choosing exhibiting and non-exhibiting tests for Software Reconnaissance, we wanted to show that Software Reconnaissance breaks down in the situation described in Claim 2: when creating surgically precise test suites is not a possibility (due to lack of experience with the target system), and instead, a previously available test suite is used in its place.

Table 4.3 compares the results of the two techniques for the five features analyzed. The table shows that the Software Reconnaissance technique found no interesting methods in 7 out of 10 trials. Only 3 trials found any, and they were all in expanded sets. These trials found only a few of the interesting methods but also a significant number of false positives.

Although the Software Reconnaissance results of trial RFE638311 Expanded return 7 elements, 3 of which are actually relevant (a 42% success rate), it is an overall fewer number of relevant elements than contained in the DFT. The number of relevant elements discovered in the DFT is 20% of the top 50 elements, or 25% of the total number of relevant elements (versus 7.5% of total elements found by Software Reconnaissance). On top of this, DFTs aid the developer in sifting through the ranked elements by allowing the developer to view the context that each ranked method was executed in the feature’s test set using the calls. As described next in Section 4.3, this information can help a developer quickly sift through the ranked methods to determine the relevant ones. Software Reconnaissance, however, does not make this information available.

\(^3\)Software Reconnaissance is typically used at the block granularity, not the method granularity.
Table 4.3: Comparison of DFT results to a Software Reconnaissance approach.

Although we did not perform the Software Reconnaissance at the block level, we have shown that the technique, performed at method level granularity, discovers fewer relevant code elements than DFTs do.

4.3 Case Studies

In this section, we discuss Claims 3 and 4 by applying DFTs to perform feature enhancements on two open source systems. We will show how the information stored in a DFT is not readily available from standard sources such as Eclipse search functionality.

4.3.1 Task 1: Cookie Enhancement

This task was to mimic a feature enhancement on the HTTPUnit codebase, corresponding to a CVS check-in on December 12, 2002. The enhancement consists of adding the ability to get and set cookies using the Javascript `document.cookie` statement. The task was carried out by the author of this dissertation who had little knowledge of the code base prior to starting the task. Although the task did not entail a significant amount of code being written or changed, it did involve learning how Javascript is handled in HTTPUnit.

Performing The Task

After reading the relevant newsgroup thread regarding this enhancement, we realized that the enhancement consisted of two parts, corresponding to two features:
manipulating cookies, and calling statements on the Javascript document object.

We first looked at how cookies are manipulated, and spent about 10 minutes to choose two test cases that seemed appropriate for testing general cookie functionality, mapping them to a feature that we called Cookies.

After running the test suite, collecting the data (with all heuristics weighted evenly), and importing the data into JQuery, we explored the feature trace starting from the highest rank. We soon discovered methods for manipulating cookies including addCookie() and removeCookie().

With a basic understanding of how cookies are manipulated, we shifted our focus to learning how to get and set Javascript object properties. We started by writing two partially implemented test cases that tested the reading of document.cookie and the writing to document.cookie, and added these two tests to a new feature that we called DocumentGetSet. We also added a third test to the feature that called the document.links property, which retrieves all the links of a web page. This third test case has a similar structure to the other two test cases and thus provided a template for how to perform the task.

We ran the test suite and, as expected, the two new tests failed. However, they still contributed code to the execution trace of the feature. Again, the data was imported into JQuery and we queried the feature trace. Two methods: postAlert(), and getNextAlert(), both within the top ten of the returned results, caught our attention. We performed a backwards trace from postAlert() and eventually found the method JavaScriptEngine.performEvent() which also had a high ranking in the execution trace.

We then performed a forward trace from performEvent(). After examining the methods returned by the forward trace, we noticed that the DFT tool results included a call from performEvent() to several other methods in the Document class including jsGet_link(). However, a visual inspection of the code showed no such calls. At first we thought it was an error in the tool, but on closer inspection, we realized that the performEvent() did in fact call these methods, albeit indirectly via a call-back mechanism through library code. Regardless, jsGet_link() seemed interesting because one of the test cases included in the feature used the Javascript statement document.link. Thus, the dynamic call graph of the DFT helped us view code relationships that do not exist statically.

With some idea of how to make the change, we added two methods stubs to the Document class: jsGet_cookie() and jsSet_cookie(). After re-running the tests, we determined that the new methods were indeed called at the expected place of the execution.

The next step was to determine how to give these two new methods access to
the document's cookies. Earlier in the task, we had determined that the `WebClient` class maintained the cookies in a `CookieJar` object, but in the `Document` class there was no direct access to the cookies. By performing a forward stack trace query on the `WebClient` constructor, we determined that the constructor transitively calls `Document.initialize()`. We refactored the necessary methods so that the `WebClient` passed its `CookieJar` as an argument to the `initialize()` method (refactoring some intermediary methods as well).

Finally, since we could now access the cookies in the `Document` class we filled in the code for the `jsGet_cookie()` and `jsSet_cookie()` to get and set cookies in the manner prescribed in the newsgroup. We then ran the test suite one last time, producing no errors or failures, thus completing the change. We changed or added a total of eight methods in about two hours.

**Analysis**

After using the ranking to choose some places to start looking, the dynamic call graph became a useful tool in helping us understand the ways in which the interesting methods were used in each test set.

Exploring the rankings top down was effective to find initial seeds to start exploring in more detail. However, the overall ranks were not decisive, i.e. not all the top-ranked methods were immediately relevant and, conversely, not all relevant methods were top-ranked. Despite this, we were nevertheless able to complete the task since many relevant methods were identified and ranked highly.

Next, we illustrate two instances in this task where we used information contained in DFTs not readily available from standard Eclipse search tools.

**Looking for Cookies** We commenced the task by seeking to understand how cookies are implemented. Had we not created a DFT to help us with this, we could have used Eclipse search instead, but we would have had to wade through more irrelevant code. A case-insensitive search for declarations of type `*cookie*` returns 17 matches, 6 of them are in the HTTPUnit codebase and the other 11 are declared in libraries used by HTTPUnit. A case-insensitive search of the methods `*cookie*` yields 113 results, with 65 declared within HTTPUnit code. It would have been possible to search through this, but would have been time consuming.

Compare this to the rankings from DFTs, which ranked two methods (`addCookie` and `removeCookie`) in the top ten. From this we were immediately able to narrow our search to those two methods and explore call hierarchies from there.
Getting and setting Properties in document Object  Using the DFT, we were able to discover that properties and functions in Javascript are *get*, *set*, or *executed* by reflectively calling the entity of that name. For example, a page's links are *get* and *set* by calling the methods `jsGet_link` and `jsSet_link` on the `Document` class. Whereas, the `Document` class could have been found quite simply by using Eclipse search, the mechanism for reflectively calling Javascript properties could not have been easily understood using only standard Eclipse functionality.

Exploring the static call graph would not be effective since the `jsSet_xxx` and `jsGet_xxx` methods are not called directly, but rather reflectively via a callback mechanism through library code.

The Eclipse call graph view for `performEvent()` is shown in Figure 4.1. This is the method that we had explored in the DFT to learn about how Javascript properties were manipulated. However, in this view none of the calls emanating from `performEvent()` in Figure 4.1 show anything in the HTTPUnit code base. Rather it is all library code, which then calls back to HTTPUnit code (not shown in a static call graph). Recall that we were unfamiliar with the system and had no knowledge that a callback mechanism was used at all.4

Exploring the dynamic call graph stored in the DFT for the same `performEvent()` method in Figure 4.2, it is much easier to see the dynamic structure and understand how the method accesses Javascript elements. Notice how the DFT does not show the library code, but rather just shows the methods *as they were executed in the context of the feature*.5 Regardless, there is no way to view the reflective calls involved with the feature using static techniques alone.

Using the DFT dynamic call graph justified why the methods `jsGet_links` and `jsGet_forms` were included in the feature. We were able to use this knowledge to implement cookies in the same manner.

### 4.3.2 Task 2: Table Creation

In this task, the author of this dissertation took the stance of a developer new to the Axion system who wants to understand how tables are created for a database. We show how DFTs can be helpful for developers trying to understand an existing feature. This task does not extend a feature, but rather shows how DFTs capture a feature's implementation. Again, we make comparisons to Eclipse search functionality to show that the same information is not readily available using

---

4Note that we did not choose this task because it relied upon reflection, in fact we did not know that any reflection existed in the feature until we started exploring.

5It would be possible to include the library code in the DFT, but in order to do this the library must be woven with the DFT analysis engine.
Figure 4.1: The static call graph emanating from `performEvent`.

standard tools.

A quick search through the development site's mailing list archive and issue tracker shows no posts dealing directly with table creation. And there was no direct access to the developers themselves. The only place to find this piece of information out is from the code itself.

We imported the Axion project into the Eclipse IDE and performed a few
searches through the Axion code using standard Eclipse search functionality. A search for all declarations of a type called "Table" yielded 107 results. Most of them have nothing to do with table creation. Although table creation classes could have been discovered by searching through every single one of those 107 classes, it would be difficult to understand how table creation is performed by browsing only the static relationships between code artifacts and it would be tedious and time consuming to explore all of them.

We decided used DFTs to help with this task. The Axion database project has 1,453 test methods in 265 test classes. A search of "Test Table" yielded 8 results (all test classes) and we quickly filtered it down into two of interest: TestTable and TestCreateTableCommand. After reading the comments at the top of the two test classes, we decided that TestCreateTableCommand directly tests table creation, whereas TestTable tests general table functionality. Thus, we found that it was eas-
ier to understand the purpose of a test class than it was to determine the purpose of a standard class.

After partitioning the test suite into a test set corresponding to one feature called Table Creation and all other tests in the default partitioning, we created the DFT using the DFT tool and imported the results into JQuery.

From here, we saw that the five most important for the Table Creation feature are: CreateTableCommand, createTable, getTableFactory, addTable, and execute. We then explored each individual method and its call stack when it was executed by the test methods in the Table Creation feature. Using the DFT, we were able to understand the steps that the Axion database takes when it creates tables.

Had there been no access to the DFT dynamic call graph, the task still could have been completed, but there would have been more data to wade through. For example, in the static call graph (from the Eclipse Call Graph view), the execute() method in class CreateTableCommand is called by 31 unique methods throughout the entire Axion system, creating tables for many different specific purposes. However, the dynamic call graph has only 2 incoming calls, both directly from the test cases in the test set which illustrate in a direct manner the standard way that execute is called when creating tables.

The time spent on the task was 5 minutes for partitioning and 30 minutes for exploration.

4.3.3 Task 3: Column Metadata

This task explores how DFTs can be used to help with a simple extension of an existing feature.

Issue 8 in Axion's bug tracker asks for more comprehensive column metadata. Before issue 8 was completed, only some column metadata was accessible from a result set, e.g. column name and column type. Issue 8 asks for the addition of precision and scale metadata for numeric typed columns.

Since this issue asks to expand an existing feature, one reasonable first step was to understand how the feature was originally implemented. Partitioning the test suite based on a Column Metadata feature and creating a DFT from that partitioning, will highlight exactly how column metadata is retrieved from a result set. Other metadata can be extracted in a similar way.

The author of this dissertation, largely unfamiliar with Axion read the details of Issue 8 and discovered that the column metadata feature is implemented so that only some metadata is accessible programmatically (other metadata exist, but it is not accessible). A search of the test case for “test*Metadata” finds 6 test
methods that test metadata directly, but only two of them test column metadata. We partitioned the two column metadata tests into the Column Metadata partition and put the remaining 4 tests into the EXCLUDE partition (so they would have no affect on the DFT creation). After executing the test suite, we viewed the DFT data in JQuery.

We saw that of the top 5 ranked methods for the Column Metadata feature, three of them are: getColumnType, getColumnName, and getDataType, which all retrieve some piece of the metadata for a column. By exploring the methods themselves as well as the call stack that led to the execution of these methods, we were able to determine the existing means of extracting column metadata.

Understanding this, we then implemented accessor methods to the other metadata specified in Issue 8 by closely mimicking the existing metadata accessor functions.

Partitioning and performing the change took less than 40 minutes combined.

4.4 Re-weighting the Ranking Heuristics

This section discusses a study we performed to see if there was a way to combine the heuristics to increase the number of interesting methods clustered in the top ranks. Originally, the three heuristics were weighted equally, but we did not believe this was optimal. This experiment was exploratory and does not substantiate a claim.

Varying the weightings of the three heuristics, we maximized the number of interesting elements ranked in the top 50, linearly scaling the elements so that an element ranked first would count 5 times more than an element ranked fiftieth. The intuition is that methods that are higher ranked should be weighted more heavily since they are more easily recognizable. Table 4.4 shows the weightings that produced the strongest cluster in the top ranking.

Our results show that there were significant variations between the optimized weightings of the features. We could not find a single optimal weighting that would work for all features that we studied. However, because the average weighting for Specialization is more than %50 greater than the averages for the other two heuristics, it seems that Specialization is the strongest of the three. Also, since the Call Depth is 0.00 in 6 of the 10 features, it seems that the heuristic as it is currently implemented is not a strong indicator of a method's relationship to a feature. This suggests that in future studies, we should weight Specialization greater than the other heuristics.

Another interesting observation is that the feature RFE 638311 Core only has one method and no combination of weightings could push it up to the top 50
ranks, therefore we were unable to compute an optimal weighting for this case.

### 4.5 What did we show?

In this chapter, explored 4 claims about DFTs.

Section 4.1 describes an experiment that tested how effectively the heuristics cluster methods relevant to a feature in the upper ranks. It showed that the heuristics did indeed perform some clustering, but that they were not effective in all situations. Also, it showed that these DFTs could be created with little prior knowledge about the target system. However, we did show that if test suites are partitioned with extreme care, then the rankings can be more effective.

Section 4.2 describes a comparison between Software Reconnaissance and DFTs, showing how a developer unfamiliar with a system can create DFTs that more closely capture the implementation of a feature than can be achieved by applying Software Reconnaissance to the same system.

Section 4.3 showed how DFTs capture knowledge about features that are not readily accessible in standard tools, in particular, Eclipse search. It also showed how DFTs capture why a method is part of a feature.

Lastly, in Section 4.4, we described an exploratory study which looked at the
efficacy of the heuristics relative to each other.
Chapter 5

Conclusions

In the thesis statement, we claimed that it is possible for a developer unfamiliar with a software system to use existing functional tests to identify source code relevant to those features as well as to understand why the source code is relevant.

We introduced the concept of Dynamic Feature Traces (DFTs) to validate the claim. We have described heuristics that rank methods meaningfully for the task at hand. These heuristics make a determination to what extent a method is relevant to a feature. However, there is still work that needs to be done in fine-tuning these heuristics. We have also shown that since DFTs retain some part of each method's calling context for a feature, a developer can more easily deduce why a method is important to a feature.

Using a comparison to the Software Reconnaissance technique, we have also shown how DFTs can provide more relevant information than other feature location techniques when the target system is unfamiliar.

5.1 Contributions

Using DFTs for feature location improves upon existing techniques by doing all of the following:

- they do not require significant knowledge of the target system in order to be useful since they are created using existing test suites,

- they maintain a concept of to what extent code is part of a feature by ranking methods executed by a feature's test set in order of relevant (rather than retaining only some code, but discarding potentially useful information, or retaining all code, but in an unstructured format) and,
• they retain some of the calling context for each method call occurring during a test set’s execution, thus helping a developer understand why methods are part of a feature.

Although DFTs cannot perfectly characterize a feature, information relevant to a feature is more accessible to a developer using DFTs than to a developer using only standard search techniques.

5.2 Future Directions

The relationships between different types of artifacts from different stages of the lifecycle or models of the system are of interest to us. We believe that work needs to be done in exploring how to find these relationships between heterogeneous artifacts, and how, once found, they should be represented. Also, we want to explore how these relationships can be explored to the benefit of a development team.

The largest area of future work directly related to DFTs is with the ranking heuristics. We have identified, but have not yet implemented several heuristics that may be useful in finding relationships between features and code. Also, there is much work in fine tuning the existing heuristics.

The sample size of our quantitative evaluation was small. In order to obtain statistically relevant results, DFTs would need to be applied to many more features over a wide variety of systems. We could see how the quality of the rankings change as the systems change, and potentially be able to categorize different types of systems based on this. Also, we would be able to see how this technique could be used in an actual development environment for an ongoing project.

A user study where some users have access to a DFT and others do not, but they all must expand the same feature, would help evaluate DFT’s overall usefulness.

5.3 Concluding Remarks

For many software systems, especially those built using a Test Driven Development strategy, the test suite is the only form of documentation that does not become stale as the system evolves. For these types of systems, therefore, the only reliable source of information about them is often the test suite itself (even an original developer’s understanding of a system cannot necessarily be trusted since it deteriorates over time). This dissertation has shown a novel technique to leverage a system’s test suites in order to assist developers who have no other reliable source of information.
Bibliography


54


