FEATURE LOCATION USING UNIT TEST COVERAGE IN AN AGILE DEVELOPMENT ENVIRONMENT

A dissertation submitted to Kent State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

by

Gregory S. DeLozier

August 2014
Dissertation written by
Gregory S. DeLozier

B.S., University of California, Riverside 1982
B.S., University of California, Riverside 1985
M.S., Kent State University, 1991
Ph.D., Kent State University, 2014

Approved by

Jonathan I. Maletic, Chair, Doctoral Dissertation Committee
L. Gwenn Volkert, Members, Doctoral Dissertation Committee
Robert Walker
Denise Bedford
Eugene C. Gartland

Accepted by
Javed I. Kahn, Chair, Department of Computer Science
James L. Blank, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES .................................................................................................................. IX

LIST OF TABLES ...................................................................................................................... XI

DEDICATION .......................................................................................................................... XII

ACKNOWLEDGEMENTS ......................................................................................................... XIII

CHAPTER 1 INTRODUCTION ................................................................................................. 1

1.1 Motivation and Relevance .............................................................................................. 2

1.1.1 Development Benefits .............................................................................................. 4

1.1.2 Academic Benefits .................................................................................................. 5

1.2 Problem Statement ......................................................................................................... 6

1.2.1 Rationale ................................................................................................................ 8

1.3 Research Questions ....................................................................................................... 13

1.4 Contributions of this Work .......................................................................................... 14

1.4.1 Modeling Agile Artifacts to Support Feature Location ............................................. 14

1.4.2 Verifying Method Correctness ................................................................................. 15

1.4.3 Experimentally Evaluating Method Effectiveness .................................................... 16

1.5 Organization .................................................................................................................. 17

CHAPTER 2 BACKGROUND AND RELATED WORK ..................................................... 19

2.1 Feature Location ............................................................................................................ 19

2.1.1 Static Analysis Methods ......................................................................................... 19

2.1.2 Dynamic Analysis Methods .................................................................................. 25

2.1.3 Feature Location Survey ....................................................................................... 32
2.2 Test Selection Methodologies ................................................................. 33
   2.2.1 Test Creation for Rapid Development ............................................... 33
   2.2.2 Test Selection for Regression Testing ............................................... 33
   2.2.3 Test Selection for Debugging ........................................................... 34
2.3 Code Coverage ......................................................................................... 34
2.4 Agile Methodologies ................................................................................. 35
   2.4.1 Philosophy of Agile Development ...................................................... 35
   2.4.2 Early Agile Processes ......................................................................... 37
   2.4.3 Agile Practices ................................................................................... 39
CHAPTER 3 A MODEL FOR EXTRACTION OF AGILE FEATURES ............... 45
3.1 Agile Artifacts ........................................................................................... 45
   3.1.1 Feature Descriptions .......................................................................... 45
   3.1.2 Iteration Plans .................................................................................... 46
   3.1.3 Source Code Repositories ................................................................. 46
3.2 Previous Models ....................................................................................... 47
3.3 Definitions ............................................................................................... 48
3.4 Modeling Agile Practices .......................................................................... 49
3.5 Identifying the Invoking Test .................................................................. 50
3.6 Identifying the Excluding Test .................................................................. 50
3.7 Relating the Model to Agile Artifacts ....................................................... 52
3.8 Identification of Containing Tests ............................................................. 53
3.9 Creation of Excluding Coverage Sets ....................................................... 55
3.10 Risks and Concerns .......................................................... 56

CHAPTER 4 PILOT INVESTIGATIONS ........................................... 58

4.1 Prototyping Data Collection Systems ........................................ 58
  4.1.1 Code Coverage in ANSI C Using Code Rewrite ......................... 59
  4.1.2 Code Coverage in Python Using Trace Interception .................... 60

4.2 Simulating Iterative Development and Feature Extraction ................. 61

4.3 Pilot Feature Extraction Results ............................................. 64
  4.3.1 First Iteration .......................................................... 64
  4.3.2 Second Iteration ...................................................... 65
  4.3.3 Preparatory Refactoring .............................................. 67
  4.3.4 Third Iteration ........................................................ 69

4.4 Pilot Feature Extraction Conclusions ........................................ 72

4.5 Designing the Feature Viewer Application .................................... 73
  4.5.1 Existing Interfaces for Code Browsing ................................ 73
  4.5.2 Prototyping the Feature Viewing Capability in the Viewer .......... 76
  4.5.3 Pilot Implementation of the Feature Viewer ............................ 78
  4.5.4 Pilot Experiment with the Feature Viewer .............................. 81
  4.5.5 Final Implementation of the Feature Viewer ............................ 83
  4.5.6 Participant Behavior Observations .................................... 90

4.6 Pilot Location Usability Study Results ..................................... 91

CHAPTER 5 EXPERIMENTAL METHODOLOGY ............................... 94

5.1 Experimental Questions ..................................................... 94
5.2 Available Resources

5.3 Preparatory Activities 

5.3.1 Identification of Suitable Code to be Searched
5.3.2 Extraction of Agile Artifacts
5.3.3 Extraction of Unit Test Feature Locations
5.3.4 Identification of Features to be Located
5.3.5 Preparation of Experimental Workstation and Environment
5.3.6 Creation of Experiment Worksheets

5.4 Experimental Participants

5.5 Experimental Protocol

5.5.1 Session Initiation
5.5.2 Session Data Collection
5.5.3 Session Conclusion

5.6 Data Processing

5.6.1 Data Assembly
5.6.2 Computation of Normalized Times
5.6.3 Location Quality Score

5.7 Evaluation of Results

5.8 Addressing Threats to Validity

CHAPTER 6 RESULTS

6.1 Feature Extraction
6.2 Experimental Evaluation of Location Usability
A.3 Experimental Study Software ........................................................................ 165

APPENDIX B EXPERIMENTAL INSTRUMENT ............................................. 166
APPENDIX C ACCEPTED FEATURE LOCATIONS........................................... 167
REFERENCES..................................................................................................... 169
LIST OF FIGURES

Figure 1.1. Feature location with ideal excluding trace................................................. 9
Figure 1.2. Incomplete excluding trace, producing incorrect feature location ................. 9
Figure 1.3. Feature location with additional code executed in the excluding trace........ 10
Figure 1.4. Feature location with union of excluding traces........................................ 11
Figure 2.1 The test driven development (TDD) workflow ........................................... 41
Figure 3.1. Sequence for processing code coverage in iterations.................................... 55
Figure 4.1. The Python tracing function used with settrace ......................................... 61
Figure 4.2. Example feature/story test code...................................................................... 63
Figure 4.3. Source code with trace information after second iteration.......................... 65
Figure 4.4. Source code with trace information after refactoring.................................... 68
Figure 4.5. Source code with trace information after third iteration.............................. 71
Figure 4.6: Code browsing in OSX Code Runner .......................................................... 74
Figure 4.7: Code browsing in Python Scripter ................................................................. 75
Figure 4.8: Code browsing in Visual Studio 2010 .......................................................... 76
Figure 4.9: Prototype of code browsing by feature in viewer........................................ 77
Figure 4.10: Prototype of feature browsing with user-identified features....................... 78
Figure 4.11: First working implementation of the feature viewer .................................... 80
Figure 4.12: Final production implementation of the feature viewer............................. 83
Figure 4.13: Searching for keyword in the feature test list............................................ 85
Figure 4.14: Searching for multiple keywords in the feature test list............................... 86
Figure 4.15: Adding exclusion keywords to the feature test list................................. 87
Figure 4.16: Basic text searching in the source code window................................. 88
Figure 4.17: Saving the feature location file................................................................. 89
Figure 5.1: Feature extraction workflow................................................................. 99
Figure 5.2: Experimental session workflow............................................................. 105
Figure 5.3: Line selection occurrences for feature C................................................. 109
Figure 5.4: Accepted location line set for feature C.................................................. 110
Figure 5.5: Computing the location quality score...................................................... 111
Figure 5.6: Data analysis and evaluation workflow.................................................... 112
Figure 6.1: Time-to-locate result histogram.............................................................. 121
Figure 6.2: Confidence-in-location result histogram.................................................. 122
Figure 6.3: Location quality score result histogram.................................................. 123
Figure 6.4: Time vs. confidence for control and experimental groups...................... 124
Figure 6.5: Visualization of selected locations for control and experimental groups... 125
Figure 6.6: Quality score vs. confidence for control and experimental groups.......... 126
LIST OF TABLES

Table 4.1. Simulated iterations ................................................................. 62
Table 4.2. Issues identified in focus groups and resulting modifications............. 82
Table 5.1. Tasks requested of participants during the experiment ....................... 100
Table 5.2. Example participant data records...................................................... 106
Table 5.3. Example task data records, after combining with location data .......... 107
Table 5.4. Validity threats / risks and mitigation steps taken.............................. 114
Table 6.1. Descriptive statistics for test coverage set sizes................................. 116
Table 6.2. Lines of code, non-comment code, and covered code......................... 117
Table 6.3. Depth of coverage – count of tests invoking covered lines.................. 118
Table 6.4. Participant question results ........................................................... 119
Table 6.5. Control (text search only) location task results .................................. 120
Table 6.6. Experimental (with feature tool) location task results .......................... 120
Table 6.7. Experimental hypothesis testing with independent t-test ...................... 127
Table 6.8. Experimental hypothesis testing with paired t-test ............................. 128
DEDICATION

This dissertation is dedicated to my wife, Janet, who has constantly supported and encouraged me for many years; to my children, Christopher, Katie, Jennifer, Elizabeth, and Rebecca, who have endured the odd circumstance of spending their entire childhood with their father in graduate school, and have done so with grace and good humor; to my sisters, Kimberly and Stephanie, who have followed my journey with remarkable enthusiasm; and to my parents, David and Pauline, who always, absolutely, knew I could do this. Thank you all, though words are barely enough.
ACKNOWLEDGEMENTS

I would like to acknowledge first and foremost the continuous encouragement and sound advice received from Dr. Jonathan Maletic. His endless patience, unfailing wisdom and excellent coaching are the primary reasons this dissertation was actually completed.

I would also like to thank and acknowledge the other members of my dissertation committee – Dr. Gwenn Volkert, Dr. Robert Walker, Dr. Denise Bedford, and Dr. Eugene C. Gartland – for their time and thoughtful insights. Their efforts are appreciated.

I am also grateful for the support and considerate treatment I have received from the Department of Computer Science at Kent State University. Both as a student and as a teacher, it has been, and continues to be, a privilege to be part of this department.

Thanks are due to the participants in the study, without whom I would not have had a viable project. I appreciate their enthusiasm, and value their contributions.

Finally, I would like to thank my employer, Progressive Insurance, and specifically colleagues such as Paul Martin, Gino DiFranco, Gregg Smith, Scott McPherson, Joseph Blanchard, Duane Hall, and David Krew for allowing me a great deal of latitude to pursue this project while employed with Progressive. Without their cooperation and encouragement, this work would not have been possible.

Gregory S. DeLozier
August 16, 2014, Kent, Ohio
CHAPTER 1

Introduction

In software development, it is frequently necessary to work on sections of a program that implement a specific feature, perhaps to repair an implementation defect, enhance functionality, or add new capabilities. Locating this code is often difficult and error prone as the software grows older and more complex (K. H. Bennett, Rajlich, & Wilde, 2002). Transfer of experienced developers and architects means retained program knowledge is lost (Keith H. Bennett & Rajlich, 2000). Typical manual methods of feature location become unwieldy and impractical as program size grows (Keith H. Bennett & Rajlich, 2000).

Many feature location methods have been proposed, usually combining human expertise with static and dynamic code analysis. In software reconnaissance, a programmer provides a feature-invoking test that exercises the feature, and a similar feature-excluding test that avoids invoking the feature (N. Wilde, 1994). Execution traces for these tests are collected. Code executed by the first test but not by the second implements the feature. This provides a precise location if tests are chosen carefully. However, no automated method exists to select feature-excluding tests that are comprehensive enough to mask non-feature code and still not invoke the detected feature.

In current agile, iterative software development methods, software is developed in increments consisting of small numbers of features and tests that both define and verify those features. These methods have become very common in modern development
efforts, and provide a consistent software development model (Boehm, 2002; Williams, 2010). We found that useful test selection methods can be constructed for agile, iterative software development projects by using properties and artifacts of the agile process, including iteration plans, change logs, repositories, and continuous build records. By identifying a time prior to feature implementation, we can identify a body of tests excluding the feature. At the point just after implementation, we can identify a minimal set of invoking tests. Using these test selection, the method will produce accurate results without requiring interactive test selection. The first part of our research created software tools for this method and demonstrated that it is viable for large software projects.

Since the agile methodology is a process executed under varying conditions, a feature location method requiring perfect execution of the process would be of limited use. We found that this feature location method can be made sufficiently robust that it produced improvements in the quality and speed of feature location tasks even if the agile processes were followed incompletely or inaccurately. The second part of our research experimentally evaluated the effects of providing this feature location capability to programmers engaged in feature location tasks. This more accurately assessed the applicability of the method to a wide variety of development efforts.

1.1 Motivation and Relevance

Finding feature implementation code is frequently difficult and time consuming. The code may be located in one place, or it may be spread across many files and many lines of a large program. The people who originally wrote the code are frequently not available for consultation, and left little or inadequate documentation. People who have
modified the code subsequently to add functionality to features may have been unable to find the original feature implementation and may have added additional feature code elsewhere in the code body. Rearrangements and refactoring may have led to movement of part of the implementing code to other locations in the code. There are many studies discussing the reasons for this loss of organization in the code (Keith H. Bennett & Rajlich, 2000). Regardless of the cause, the phenomenon tends to clutter the initial organization of the code, reducing feature-code cohesion and obscuring implementation information.

Not all difficulties in locating feature code result from deficiencies in software maintenance activities. Many common information-hiding practices in object methodologies, such as encapsulation, delegation, inheritance, and abstract methods tend to hide the locations of implementations intentionally to present clean interfaces to users of these software modules. Additionally, code browsing tools are generally intended to browse by files or by static artifacts such as class implementations or object hierarchies and do not lend themselves to feature location activities. It is rare for software tools to provide much more than global textual searches and object browsing for feature location tasks.

Despite these difficulties, location of feature implementation code is an active area of research. The availability of accurate methods of feature location would be of significant benefit to both developers of software and to academic research interests.
1.1.1 Development Benefits

Feature location capability in commercial development creates advantages for the development team in several areas.

- Code maintenance is easier if the location of feature implementation is found rapidly and accurately. This helps focus effort immediately on areas of work for feature enhancement.

- Testing and test planning becomes easier. If the code touched by a feature is known, those areas of the code can be targeted for testing.

- Risk management is facilitated. For instance a planned change to code can be evaluated in terms of the features that are also implemented by that code. A less risky location for the change might be selected.

- If other features are known to be impacted by a change, then those features can also be included in a test suite, lessening the risk of unintended impacts to related features.

- Understanding feature dependency is essential to planning the continued evolution of a software system (Jadallah, Galster, Moussavi, & Ruhe, 2009). Accurate feature location can contribute to this dependency information.

The proposed research is attempting to find robust, widely applicable, and accurate ways to provide these benefits by using information available in common software processes.
1.1.2 Academic Benefits

Accurate feature location capabilities create additional descriptive dimensions for source code (where other dimension might be lines of code, complexity, average number of changes/line, etc.). These additional dimensions might include:

- Number of features supported for a specific line/method/component
- Density of features/unit of code
- Average size of features
- Feature motion distance during refactoring

The availability of these features creates the possibility of new research questions. For instance:

- Does code that implements many features in a small area become less reliable in terms of defects eventually reported?
- Is code that is feature-attributed easier to refactor?
- Does feature-driven test management result in lower defect rates?
- How does expected size of feature relate to estimated size of feature? Are developers better at estimating size than they are at estimating time?

The ability to apply a feature location methodology to a wide range of software development efforts increases the likelihood that both these dimensions and other experimental dimensions can be obtained for the same body of code.

We do not, however, know if the proposed methodology is effective under normal agile circumstances. There are some circumstances that may reduce the usefulness of the proposed method:
• Simultaneous development of features in an iteration may reduce focus of the invoking test by including other features simultaneously created.
• Invoking tests may be designed to test many features in one test.
• Agile applications refactor frequently – would this refactoring diminish the ability of tests to find and exclude features?
• Does the tendency of real-world agile teams to “cheat” from time to time – deviate from the process – diminish the usefulness of the method

An experimental study can examine these questions by initially implementing the proposed methods and examining performance of the method against other known methods and then measuring statistical effects on the performance of feature related tasks by developers in real-work scenarios.

1.2 Problem Statement

It is useful when maintaining a complex software system to be able to find the code in the system responsible for implementing a given feature. The method of software reconnaissance is known to be able to locate features in code by subtracting those code locations visited in a non-feature-invoking code coverage trace from those in a feature-invoking trace, where the remaining invoking-trace locations indicate the code that implements the feature (N. Wilde, 1994). Typically, those traces are generated by running tests while collecting code coverage data. In Wilde’s original method, the tests are constructed specifically for that purpose.

The agile process and methodology creates large numbers of tests to verify features as they are implemented. There may be thousands of available tests, each testing
an aspect of a developed feature. These tests are maintained and repeatedly applied through the life of the project to verify that implemented features initially work correctly and don’t break as development proceeds. However, there is no general method for extracting tests from agile test suites to create feature location test sets.

Finding a test (or set of tests) invoking a feature is trivial, since the tests are tagged with the features they are intended to test.

However, no specifically non-invoking test is written as part of the agile process, and there is no general method for selecting, from existing agile tests, a good candidate (or group of candidates) to create the non-invoking trace, as there is no general guarantee that any given test does not invoke any particular feature.

Eisenberg believes the nature of test driven development makes it a poor choice for excluding tests.

Test suites developed under TDD are a good source of exhibiting tests. However, they are not a good source of correlated non-exhibiting tests—indeed the TDD methodology does not call for the explicit construction of “tests that do not test feature X”. Because of this, we believe that techniques like Software Reconnaissance and Execution Slices cannot be used effectively with a typical test-suite developed under TDD, unless developers are willing and knowledgeable enough to develop correlated non-exhibiting tests themselves. (Eisenberg, 2005)

A method to use existing tests to create an excluding test needs to select enough tests to exclude the non-essential code from the invoking trace, but not one that executes the feature code. If such a general method existed, it could be used to create feature location traces that would be helpful in code navigation, code understanding, risk
analysis, and other software engineering tasks. Since agile methodologies are widespread in modern commercial software practices, such a method would have applicability to many current development efforts.

It is equally important that the proposed research specifically expects that these artifact relationships are incomplete and inconsistent, and the study design is intended to look at the usefulness of these methods in the presence of these imperfect data.

1.2.1 Rationale

It is well understood that the location of a feature implementation in a software system can be found by creating two sets of code execution traces, one for a use case that includes invoking the feature, and one for a use case that does everything the first case does except for invoking the feature. After locations executed in the second trace are removed from the first, remaining locations in the trace are assumed to execute the feature of interest.

Ideally, the trace not invoking the feature would execute exactly the code of the invoking trace, except for omitting the feature of interest. This cleanly defines the feature location, as shown in Figure 1.1.
If the non-invoking test code does not cover the entire non-feature range of the invoking test, additional code locations can be erroneously reported as in the feature code, as shown in Figure 1.2. To report features accurately, the non-invoking test must execute at least all the code that the invoking test does, except in the area of the feature of interest.

Figure 1.1. Feature location with ideal excluding trace

Figure 1.2. Incomplete excluding trace, producing incorrect feature location
It is important to note that as long as the excluding code does not execute the feature of interest, additional code executed is harmless (other than, perhaps, increasing the time required to execute the test.) This is the case shown in Figure 1.3.

![Feature location diagram](image)

**Figure 1.3. Feature location with additional code executed in the excluding trace**

Finally, it is possible that the excluding trace (and, perhaps, the invoking trace) are created from the union of more than once test trace. If there does not exist one single excluding trace, then a set of traces that collectively create the necessary coverage while not invoking the feature may serve just as well. Figure 1.4 shows such a situation.
This last possibility is especially useful, because in the unit test assets of an agile project, for each feature implemented, there will be one or more unit tests that invoke that feature. Finding an invoking test is straightforward. However, agile practice doesn't specify writing tests that specifically don't invoke a particular feature, so there won't be a ready-made exclusion test. However, it is possible to create a test from the union of many tests, where the tests are known not to invoke the feature of interest.

Here some specific characteristics of agile development become important. An methodology is agile for our purposes if it follows the following rules:

- All features are developed in periods called iterations, where before the iteration, the feature does not exist, and at the end of the iteration, the feature is fully functional.
- Each feature is defined and verified by a collection of one or more unit tests, which fail before the feature exists and pass after the feature is
completed, and cover (i.e. invoke) all of the code that implements the feature.

From these rules, we can make the two following observations:

• The code coverage intersection of all tests that exist immediately before the start of an iteration cover all features created before the start of that iteration. In other words, this intersection can be used to exclude all features in existence before the iteration.

• A test designed to verify a feature can only do so by calling that feature in the context of all other features created before the iterations started. The feature test cannot depend on other features not yet completed to set up the feature test.

These properties of agile artifacts will be discussed more formally in Chapter 5.

Unfortunately, just looking at the changing code body over time does not allow trivial connection between code instance information and iteration data. Just by looking at check-ins and builds over time, it isn't generally possible to identify the time at which a feature appeared, nor the timeline of the containing iteration.

To enable this, we can mine the management artifacts of the agile process. Story cards contain informal descriptions of features to be developed, and these are generally maintained in a store that contains both the feature data and the iteration it was assigned to. Additionally, records of the iterations can be used to determine the beginning and end of the iteration of interest. Those dates and times will help to locate points in time on the
development chain where the test collection will have the qualities outlined in the two observations above.

In summary, we will use management data and an understanding of the agile process to determine chronological boundaries for feature development, and source data to extract coverage-creating tests that isolate those features.

1.3 **Research Questions**

We wanted to examine common artifacts of the agile development process that have known relationships to both the tests and the code under test, such as story cards, test records, and source repositories, and determine if a partial ordering of the tests using these artifacts can be used to determine a useful set of tests for the non-invoking collection for any specific feature. A set of tests is considered useful if practitioners using traces created from these test selections, and a tool for viewing code based on trace coloring, can locate and identify specific feature implementations in code with more accuracy or better speed than without the assistance of these traces.

Specifically, we wanted to answer the following questions:

- Where in the agile process artifacts can useful data be found that can help us relate tests to features of interest, and how does the model help us relate those data to those features?

- Can a selected set of partially ordered tests, shown by artifact evidence to not include the feature of interest, serve as a sufficiently useful non-invoking test?
• How well does such a method perform in actually locating features in code without manual intervention or “tuning” of the test selection process?

• Can developers using those tests be shown to perform better in feature identification tasks than developers using traditional manual feature location methods?

1.4 Contributions of this Work

To effectively answer the research questions, a model-based methodology and experimental evaluation are used. The contributions of this research are:

• Creations of a model of agile methodology artifacts and a method for selecting necessary tests for feature location via software reconnaissance.

• Verification of the model and method for extracting feature locations from source code and tests in agile projects via a user-based experiment

• Verification that the method is sufficiently robust to be effective in enabling the task of locating features in real-world development efforts.

This required an extensive investigation, including model and method creation, method verification on synthesized and real-world source code, and experimental evaluation of real-world performance effects. We discuss these activities in more detail.

1.4.1 Modeling Agile Artifacts to Support Feature Location

We developed a method for locating features using software coverage masking in software developed with agile practices, and identified the data in the agile method that can be used for feature location.
A simple model of the major artifacts in the agile software development methodology was defined. This model allows us to reason about the relationship between the properties of agile artifacts and the requirements of tests in the feature extraction method. In particular, we use the properties of the agile artifacts to determine how test coverage can be generated to serve as the excluding tests in the agile space. This model was sufficiently well developed when it could provide clear support for these selections.

The model also serves as an abstraction layer allowing us to quickly guide a real-world implementation of the method by mapping each model abstraction to physical artifacts in the software process. This tells us, for any real-world process, if the necessary mappings (i.e. artifacts) are in fact present and accessible. If they are, the mapping provided an architectural context for extracting the relevant data from the process.

Details of the agile artifact model can be found in Chapter 3.

1.4.2 Verifying Method Correctness

We verify that features can be successfully located using the method described apply to the data extracted via the model from agile processes. For this purpose, we created a simple, non-interactive software system that extracts agile data and identifies source code lines related to a feature.

We first evaluated results on a small agile example project against manual evaluation of feature locations. On this small project, we simulated iterations and feature sets as we created the project, so we were able to generate a number of use cases. We used these use cases to evaluate the method, and to fine-tune the method by making
additions and corrections to the model (and subsequent software) to clarify or create capabilities.

When the method correctly extracted features, in accordance with the model and predictions made from the model, we selected a well-known open source project that was created with agile methods, and used the method to extract features from that project. The extracted feature locations were compared to manually identified feature locations, and to feature locations identified by other methods. This approach is similar to the one taken by Marcus in examining features location in Mosaic (Marcus, Sergeyev, Rajlich, & Maletic, 2004).

Verification of the method correctness was considered successful if features were located in a way that seemed reasonable, matched the predictions of the model, and seemed to experienced programmers to give sufficient information to be useful for feature location. No statistical evaluation of performance was planned for this evaluation, though a summarization of performance was provided.

Further discussion of the evaluation and verification of method correctness can be found in Chapter 4.

1.4.3 Experimentally Evaluating Method Effectiveness

For our final objective, we experimentally evaluated the effectiveness of the method in real-world development. Agile development efforts don’t always follow the agile process perfectly. We wanted to find out whether the proposed feature location methods were useful, where useful meant that feature location tasks were done at higher quality or done faster, when used in agile efforts where processes weren’t followed
perfectly. To do this we performed an experiment, examining the null hypothesis that access to this feature location method had no effect on the performance of programmers engaged in feature location tasks.

We measured performance of programmers who were given tasks to locate features in real-world agile projects of significant complexity, both with and without the use of the feature location method. The method was provided as an available feature in a software source code browsing program with browsing and search capabilities similar to that found in current development environments such as Visual Studio. The subject was given a set of feature location tasks, to be completed both with and without the use of the feature location tool, using an approach similar to the with/without tool strategy used by Saff and Ernst (Saff & Ernst, 2004). We determined if the method had a measurable effect on the speed and accuracy of feature location tasks in real-world software projects. Speed was measured with an observer’s timer. Identification quality was assessed by using a completion algorithm to assess the quality of feature identifications done by subjects without regard to the method associated with the identification.

If the programmers performed significantly better on feature location tasks, we would be able to conclude that the method was sufficiently robust to be of use in real-world agile projects.

Details of the methodology, instruments, and subsequent analysis can be found in Chapter 5, in the discussion of experimental methodology.

1.5 Organization

The remainder of this dissertation is organized as follows:
• Chapter 2 discusses the history of the feature location problem, current research into feature location, and a summary of the fundamentals of agile development methods.

• Chapter 3 presents a conceptual model for extracting features by using test masking methods and artifacts derived from agile development processes.

• Chapter 4 discusses a number of pilot investigations used to refine and verify the extraction model, and the design of an end user tool designed to present the feature location capability to an end user.

• Chapter 5 discusses the methodology for the investigation, including the extraction of feature information from tests, and the experiment to assess the impact this capability has on software developers locating features.

• Chapter 6 presents the results of these investigations, including statistical analysis of the research hypotheses

• Chapter 7 discusses the results and significance of the research, and presents an overall conclusion.

• Appendices present additional helpful information: program listings, experimental worksheets, and discussion of source code availability.
CHAPTER 2

Background and Related Work

2.1 Feature Location

A number of researchers have examined the problem of finding feature-implementing code in software programs. Early efforts focused on static methods, analyzing the source code to find the locations of feature implementations. Researchers investigated how to manually search program text for the implementation location, and some tools were created to assist in these efforts. Later efforts automated static analysis of the program source code, as well as the analysis of source code history. Dynamic methods have also been investigated, examining both the tracing of software components being executed as well as the dynamic relationships between the data objects during software execution. Finally, more rigorous mathematical models such as concept analysis and latent semantic indexing have been proposed for feature location in both static and dynamic analysis scenarios.

We will review the principal works concerning static and dynamic analysis methods, paying specific attention to the potential of each method for use in building toward completely automated feature location.

2.1.1 Static Analysis Methods

In static analysis, the method looks at the program text, and derives information from that text in the context of the syntactic rules and semantic structures of the language.
The program is not actually executed; this is a passive examination of the program’s intent.

*Manual Evaluation*

Manual methods of feature location employ no specific tools to support the feature location activity. These methods describe cognitive approaches to looking at code with conventional software tools.

Before feature location was studied explicitly, there were a number of efforts to understand the activity of program comprehension in general. Brooks created a model of program comprehension that described the understanding of existing code as a process of reconstruction of layered problem mappings, driven by hypothesis and experimentation (R. Brooks, 1983). Soloway further described comprehension as a series of inquiry episodes, each consisting of reading code, questioning something about it, conjecturing an answer, and searching for confirmation (Soloway, Adelson, & Ehrlich, 1988).

In these initial understandings, the tools concerned were not particularly important. Methods described are exercised manually, using ordinarily available tools such as text editors, pattern match scanners, etc. for examining code.

Initial descriptions of specific feature location explored the various activities one might employ using standard coding tools to find feature implementations. Lakhotia described informal experiments where he observed the behavior of himself and several students while modifying large computer programs, namely the GNU C compiler (gcc) and the Wisconsin Program Integration System (wpis) from the University of Wisconsin (Lakhotia, 1993). He observed:
• Modifications could reasonably be done by understanding only the part of
the code that implemented the feature being modified.

• Expert (i.e. experienced) programmers were much more capable of rapidly
scanning a body of code looking for the parts requiring modification.

• The expert looks for clues to the location, creates theories about how the
program works, creates a mapping based on that theory, and recursively
hones the map in areas relevant to the feature, both in locating files of
interest and code within those files. The map of understanding was
increasingly (and more selectively) detailed, as the programmer was able
to “zero in on” the area to be modified.

• His observations supported the comprehension models of Brooks and
Soloway.

• Tools like simple editors and pattern scanners (i.e. grep) combined with
reasonable guesses about identifiers of interest, were sufficient to guide an
expert programmer to the eventual discovery of the relevant code sections
for modification for simple feature location.

• Eventually tools integrated with the software build processes would be
needed to assist feature location for large projects.

Sim, Clarke, and Holt also describe the widespread use of common tools like grep
for searching for feature related code, searching for identifiers and source code comments
in source files (Sim, Clarke, & Holt, 1998). By looking at the code surrounding each
finding, additional candidate search targets are identified. Eventually sufficient feature-related code is discovered to allow understanding of the feature implementation.

**Assisted Code Browsing**

Assisted code browsers provide the user of the tool some guidance, visual cue, connection following, or other capability to assist the user in manually finding the code responsible for feature implementation. These tools do not directly analyze feature location, but provide tools to allow users to do so. There are a number of such tools and approaches available.

Mens and Poll have proposed intentional code views – an approach to code viewing where views are derived using a specification language to select relevant portions of code to view by grouping together sections of code that address the same concern (Mens & Poll, 2003). Their specification language looks at declarations of identifiers, use and manipulations of identifiers, and relationships in class hierarchies when selecting code to view.

Chen and Rajlich propose a graphical approach to browsing, representing code as a connected graph of modules, where arcs represent dependencies, such as calling functions, caller functions, declarations, usages, etc. (Chen & Rajlich, 2000). The user of the tool can explore these various arcs, expanding nodes as necessary, rapidly moving between areas of the code that might contain related feature-implementation details, until the user is satisfied that all relevant details have been uncovered. Specifically, the tool creates a "procedure dependence graph" and "system dependence graph" creating an abstract system dependence graph (ADSG) consisting of call edges, data flow edges and
function and global variable nodes. The initial area of interest is displayed. "Each visit to a component expands the search graph until all the components implementing the feature or concept are displayed." The programmer directs the search by selecting one related component for expansion. Then "the programmer checks whether all components relating to the software have been found." This process automates the human edge-following activity that any programmer might do when attempting to understand the program. They use Mosaic as a test case. This was reasonably successful. Of 984 functions in the version of Mosaic they studied, their use of this method to find the components responsible for video and media playback identified 22 functions, or about 2% of Mosaic code.

Some of the concept analysis tools discussed below have significant human assistance in browsing, using concept analysis to create arcs in the browsing tools. In this model, concept analysis creates the original graph set, but human input guides the growth of the concept set until the set includes all feature-related code.

Automated Static Analysis

These static methods use automated tools to examine the static code and related artifacts to determine the code used in implementing a specific feature.

Static program slicing is an early method of static analysis that tries to produce the smallest operational program that produces the desired feature. Early methods used slicing to reduce the amount of a program that is required to be considered to debug a specific problem. Backward static methods attempt to produce a minimal program to produce a specific output. Within this slice presumably lies the desired feature (or defect). Weiser has demonstrated that the selection of an absolutely minimal set of such
statements is undecidable, but that usably small approximations can be made (Weiser, 1984). A slice can be computed by backward traversals of the program’s dependency graph with respect to all statements affecting the specific outcome for a variable \( V \). (As opposed to forward traversals, consisting of all statements affected by a previous value of the variable \( V \).) A number of variants on this approach have been suggested (Binkley & Harman, 2003). Many of these consist of more sophisticated algorithms for reducing the slice of the slice (thus better bounding the feature implementation space), for intersecting slices (also to reduce search area), and for parallel computation of slices. Dynamic variations on slicing are discussed below in the section on dynamic analysis.

Latent semantic indexing (LSI) is another approach to textual analysis of static source code useful for many applications in program understanding. Marcus and Maletic used LSI to explore static source code to find concept locations in programs (Jonathan I. Maletic & Marcus, 2000). Even with analytical extraction, the process is subject to human interpretation. “The users play a dominant part in this process; they formulate the queries and evaluate the results returned by the system.” The query language using LSI returned more useful information than regular expression queries from \textit{grep} were able to identify. Specifically, regular expression retrieval was not able to rank the value of each found instance, requiring examination of all results, or encouraging additional refinement of the search that might lead to missing relevant information. LSI was also used to find duplicate sections of code implementing similar activities, called concept clones, by analyzing the structure and intent of source code (Marcus & Maletic, 2001).
2.1.2 Dynamic Analysis Methods

In dynamic analysis, the program code is executed, generally in an instrumented environment, and characteristics of the run-time behavior of the program are collected. Edwards (Edwards, Wilde, Simmons, & Golden, 2009) refers to the four main aspects of these instrumentation systems as

- **Insertion engine** – scans code and inserts tracing code
- **Runtime engine** – the inserted code along with supporting code to record trace
- **Collection engine** – a system to collect the record traces and store them
- **Analysis engine** – tool to reformat trace data for humans or other analytic tools

The collected code traces from the instrumented systems are then examined, along with the artifacts of the program text, to assist in feature location.

*Execution Traces and Code Coverage Methods*

These methods utilize static information plus the dynamic knowledge of what statements were executed during one or more code execution sessions. These methods became popular, as efficient code instrumentation systems were made available for a variety of systems.

Wilde initially proposed software reconnaissance as a method to locate features in instrumented code (N. Wilde, 1994). In this method, two code execution traces are compared – one containing an invocation of a feature of interest, and one that omits that feature but is in other respects as similar as possible. By comparing these two traces (or
code coverage results) the area of code responsible for implementation of the feature can be determined. Wilde tested this method in a number of various efforts in systems up to about 15 KLOC (N. Wilde & Scully, 1995).

White had success in extending the technique. While the original tools were implemented for C and C++, White, et. al. found the method to be effective at locating features in Ada code, but noted that the method required special considerations for use in embedded applications and other timing-critical systems (White & Wilde, 2001). Many of the issues were related to the requirement for monitoring instrumentation and its resulting effect on critical code.

In a later study, Wilde compared several methods of feature location for a large Fortran modeling tool (N. Wilde, Buckellew, Page, Rajlich, & Pounds, 2003). The study compared Chen’s dependency graph expansion method, the manual grep searching method, and the software reconnaissance method. A team of programmers (or in some cases, a programmer) was assigned to each method, and asked to locate specific features. All three teams were able to find a specific feature, but the manual (grep) team failed to successfully locate the more complex feature.

The software reconnaissance method was extended to distributed systems by Edwards, et. al. (Edwards, Simmons, & Wilde, 2006). This effort used tracings of concurrent time intervals, either directly or via message log recovery, to determine what software components were active at specific times when a software feature was being invoked within a component of the distributed system.
Simmons tested software reconnaissance in a commercial environment using Wilde’s method implemented with additional commercial tools: *Metrowerks CodeTEST* for dynamic code coverage, *Klocwork inSight* for static analysis, and *TraceGraph* for trace differencing. They selected the *Apache* web server containing 227KLOC, as a test case. They used a test manager to make it more convenient to collect separate traces from each test -- but this only collects a trace at during a specific time interval, possibly missing involved startup or shutdown code. Tests are created or selected manually. Quite a lot of engineer involvement is required to run the system. "Case study participants went through all the steps of analyzing a scenario: designing and running test cases..." This study tested if two people identified the same feature location using these tools, and reported varied results. They emphasize their dependence on proper test selection. "This difference shows the importance of tests cases in using dynamic methods of feature location. When a first set of tests identifies a lot of marker code, it is probably best to go back and refine the tests cases to see what additional code can be excluded." (Simmons, Edwards, Wilde, Homan, & Groble, 2006).

Wong further discussed the test selection problem (Wong, Gokhale, Horgan, & Trivedi, 1999). The author observed that systems start out with high cohesion and low coupling, but as development progresses, the clear mapping between code and features starts to degrade. "Since it is impossible, in general, to identify all the invoking or excluding tests for a given P and F, a practical alternative is to run P using a small, carefully selected set of tests T with some exhibiting F and others not." There followed a discussion of selecting invoking and excluding tests, using some manual guidance of the
process. It is useful to note that the process was not viewed as fully automated. "In fact, to find the complete set of code for a given feature, one may have to use many more invoking and excluding tests, in contrast to our technique which only requires a few carefully selected tests." In other words, a few carefully selected tests weren't enough to find the complete set of code. They presented the result as a "good starting point".

Eisenberg tried to reduce the sensitivity of feature location via execution tracing to test selection issues by introducing “dynamic feature traces” – using the test to feature set mapping to create a dynamic graph of “ranks” and “calls” (Eisenberg, 2005). Once feature set mapping to tests was complete, the trace analysis and DFT creation is complete. This paper explicitly acknowledges TDD (test-driven-development) testing, which is rare. In this case, the developer creates an explicit mapping of tests executing a given feature. Then an implicit mapping is used to automatically compare exhibiting tests to all other tests to create an excluding (non-exhibiting) set. Then the system generated calls and ranks and applied heuristics found by trial-and-error:

- Multiplicity – called many times by exhibiting tests,
- Depth – shallow are more likely to be important than deep calls,
- Specialization – only called by invoking tests.

Excluding tests were not used subtractively, but rather to provide computational comparison to the invoking tests. “A technique is considered successful if it brings elements in the core set to the developer’s attention.” Eisenberg made a useful choice of case study software in Java including Axion – code that was completely developed using
test driven development and related enhancements to CVS check-ins. Code was checked out immediately before a feature set was closed.

It is notable that for an automated technique, Eisenberg’s method involves a substantial amount of manual interaction with the test set. For instance, section 4.2.1 of the cited paper describes a very manually data intensive process for the detection of one feature. Further, the evaluation of the quality of the results is basically a self-assessment of “usefulness”.

Finally, note that Eisenberg describes the unsuitability of TDD tests for use with software reconnaissance because developers in agile methods don’t generally write excluding tests, so the tests resulting from TDD efforts aren’t a good source of excluding tests.

Poshyvanyk combined LSI with dynamic execution trace methods (Poshyvanyk, Gueheneuc, Marcus, Antoniol, & Rajlich, 2007). In this method, both static (LSI) and dynamic (scenario-based execution tracing) methods are used to determine candidates for feature implementation code. The probability of a candidate being selected correctly increases if both methods agree that the candidate is a likely selection. He concludes that the combined analysis works more accurately in finding defects in features than either method used independently.

Execution Slices and Dynamic Interaction

Program slices consist of those program statements that need to remain in the program for the program to execute for a specific combination of input variables. Korel and Laski proposed the concept of dynamic slicing. They used dynamic instrumentation
to mark statements, array elements, and pointer variables that were actually used in the execution of a feature (B. Korel & Laski, 1988). Note that this differs from code coverage in its inclusion of data elements in the profiling of a specific execution session. Program slicing is also more driven by modeling than simple observation, in that inclusion of certain statements may imply the inclusion of others to make sure that the statements extracted form an executable subset of the original program.

Many variations on program slicing have been proposed within varying degrees of data model sophistication. Korel and Rilling proposed using dynamic slices for program understanding, including feature location (Bogdan Korel & Rilling, 1998). The difference in content of two dynamic traces can be used in the same way as a comparison of execution traces, for distinguishing code segments that are included in the set of code that specifically implements a feature. Hall proposed algorithms that could generate simultaneous dynamic slices from more than one intersecting test collection (Hall, 1995). De Lucia noted that these slices could be used much the same as execution traces in Wilde’s feature location methods (De Lucia, Fasolino, & Munro, 1996).

*Feature Location via Concept Analysis*

Biggerstaff defines concepts as software characteristics less definable than features, which are only definable in terms of program behavior (Biggerstaff, Mitbander, & Webster, 1993). Concepts can describe program intentions in human terms. For instance, we could define a concept of “date” and define use cases where the program does and doesn’t execute code concerning this concept.
Eisenbarth uses concept analysis to find features in source code (Eisenbarth, Koschke, & Simon, 2001). He uses scenarios – a series of program use cases -- to map features onto computational units of various granularities. Scenarios are created by hand, involving three human activities -- analyst interested in mapping, domain expert who designs the scenarios, and a user who executes the scenarios. Many scenarios are created that implement the feature in various ways. These scenarios activate computational units. Relationships between computation units are maintained in a lattice. "The basic idea is to isolate features in the concept lattice through combinations of overlapping scenarios… With the domain expert's additional knowledge of which features are invoked by a scenario, we can identify the computational units relevant to a specific feature."

The intersection of computational units by running multiple scenarios is critical. "Computational units specific to feature f1 can be found in the intersection of the executed computational units of the two scenarios s1 and s2 because f1 is invoked for s1 and s2… Since s1 and s2 do not share any other feature.” His method also requires identification of “general-purpose building blocks" so as not to include those in the feature. He further asserts that relationships between features cannot be discovered by Wilde's recon method or by Wong's slices, but can be found in the concept lattice.

Interestingly, in this approach, human interaction and guidance is very much expected, both by manually removing uninteresting general-purpose computational units and by adding units known to be relevant but missed by the initial automated analysis.

Case studies were included in this study, including very complex and large programs.
Eisenbarth clarifies these concepts in a later paper, noting that “by considering several execution traces for different features at a time, components not specific to a feature will 'sink' in the concept lattice, i.e. will be closer to the bottom element.” (Eisenbarth, Koschke, & Simon, 2003).

Tilley et al. review 42 papers about formal concept analysis (FCA) supporting software engineering activities (Tilley, Cole, Becker, & Eklund, 2005). While not directly about feature location, the study is interesting in that it examines the success of a related technique over a large variety of environments and project sizes. In reviewing these papers, he makes note of the size of the tested program. Of the 42 papers, only 7 study projects > 100KLOC, and only two studies examine projects > 200KLOC. There were 34 projects studied (sometimes more than one to a paper) under 100KLOC, most of which were under 10KLOC. The paper is actually a meta-study, actually using the papers themselves, citation records, etc. as input data.

### 2.1.3 Feature Location Survey

A recent, comprehensive survey of feature location methodologies was presented by Dit et al. 89 articles from 25 venues were surveyed to create a taxonomy of feature location methodologies (Dit, Revelle, Gethers, & Poshylvanyk, 2013). While mentioning many of the methods discussed in here, no mention of new methodologies for feature location with masked tests or software reconnaissance was made. The use of agile artifacts was also not discussed.
2.2 Test Selection Methodologies

There are a number of circumstances where a partial ordering or binary classification is made on a group of tests for the purpose of selecting a test or group of tests most useful for a particular purpose.

2.2.1 Test Creation for Rapid Development

Test-driven development, as discussed above, requires writing of feature-defining tests before implementing the actual test. Parrish developed a more optimal test-creation ordering, moving from fine-grained, specific tests to more general coarse-grained tests (Parrish, Jones, & Dixon, 2002). This bottom-up test creation ordering matches a common coding pattern, and if test creation order is recorded, provides a partial ordering (assuming more than one developer) of tests that could be used to extract more or less specific tests regarding a feature.

2.2.2 Test Selection for Regression Testing

Leon and Podgurski compare four methods for selecting tests from large test suites adequate for testing purposes (Leon & Podgurski, 2003). Their principal concern is selecting tests that are adequate for code coverage purposes, omitting subtests that may test code already tested by a larger, more comprehensive test. The motivation to omit tests is largely practical – concerns of performance and automated test capabilities in systems with a lot of tests. This work is interesting as it uses code coverage results to determine test overlap and to assist in the reduction decisions.
2.2.3 Test Selection for Debugging

Gälli uses partial test ordering via coverage comparison to select the most specific test that displays a defect for debugging, given a defect that causes many tests to malfunction (Gälli, Nierstrasz, & Wuyts, 2003). This is done via test hierarchy evaluation, and does not use at coverage differences for this purpose.

Zeller discusses delta debugging, outlining a method for applying successive changes in a system (e.g. $C_0\ldots C_n$) to rapidly determine the change responsible for a defective behavior (Zeller, 1999). Zeller’s method is interesting in that it can be optimized using binary tree search on partitions of the change tree, so that performance is better than simply running all tests and combinations to find the bug-inducing change. Zeller’s method is also of note in using a model of the testing procedure similar to the one used in our project.

2.3 Code Coverage

Much has been written on the definition of code coverage, methods for measuring code coverage, the value of high code coverage as evidence for adequate testing, and so forth. For instance, instrumentation is covered well by Tikir and Hollingsworth (Tikir & Hollingsworth, 2002). Chockler discusses the relationship of code coverage to formal verification, defining in the process many kinds of code coverage metrics in an extensive model of coverage (Chockler, Kupferman, & Vardi, 2003). Edwards, as mentioned in section 3.1.2, presents a conceptual model for coverage instrumentation that we will use for our coverage tools (Edwards et al., 2009).
For purposes of this effort, code coverage as understood in Wilde’s software reconnaissance method – i.e. the identification of each line of code executed in the course of a running a test – is sufficient for our purposes, and examination of fine nuances of code coverage use will be left for another time.

2.4 Agile Methodologies

The techniques developed in this work will use artifacts from agile software methodologies. Here we will briefly discuss agile methods, major practices common to the majority of agile methods, and specific artifacts that we will be using for information mining

2.4.1 Philosophy of Agile Development

It has long been recognized that it is difficult to establish high levels of predictability in software development activities. Many explanations for this phenomenon have been offered, ranging from difficulties in requirements analysis to flawed coding methodologies. Detailed analysis of this difficulty has led to models that imply that predictability is not possible in current methods of software development (Lewis, 2001).

Frustration with unpredictable software activities and their high rate of failure has led to a number of attempts to define software methodologies to reduce the impact of various aspects of the problem. These attempts have addressed various sources of unpredictability by creating planning methods, software tools, debugging methodologies, testing strategies, and so forth. Brooks surveyed a number of “magic bullets” and
concluded that no single method exists that significantly reduces these problems (F. P. Brooks, 1987).

In particular, change in software requirements has been seen as a major contributor to software unpredictability and eventual failure of software projects (Ikram & Ramzan, 2007). Many attempts have been made to increase initial quality of requirements, trace requirements into software implementations, and subsequently control change in requirements (Lavazza & Valetto, 2000).

Brooks described the pressure for change well:

All successful software gets changed. Two processes are at work. First, as a software product is found to be useful, people try it in new cases at the edge of or beyond the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new uses for it.

Second, successful software survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, then at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicles of opportunity.

In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product. (F. P. Brooks, 1987)

Researchers eventually arrived at the conclusion that there is an inherent software uncertainty principle in software development that maintains that the very act of creating a non-trivial software product creates an environment where predictability is impossible to obtain (Ziv, Richardson, & Klösch, 1996). Simply stated, the act of providing a user with a significant software tool changes the environment of the user, and the new environment, the user will have different requirements.
Agile methodologies evolved under the assumption that since change in requirements is unavoidable, it is better to design a method for writing software that is amenable to frequent changes in the specification of the software. If the method is tolerant of these changes, then when they happen as expected, instead of acting as an impediment, they are treated as a normal occurrence in the development path (Fowler, 2005).

To achieve this goal of reliable software development under conditions of change, agile systems, while varying widely in details, adopt some consistent practices and artifacts that we will describe in this chapter. It is worthwhile to note that agile practices can also involve a number of aspects that we will not discuss here, as we restrain ourselves to those aspects relevant to the current effort.

2.4.2 Early Agile Processes

Many current agile methods have been practiced, individually and collectively, in various environments for quite some time. Incremental software development, in particular, has long been recommended. In the 1970s, Mills suggested “a sequence of intermediate systems” where “each system can be verified to be correct.” (Mills, 1971). Brooks wrote at length about incremental development being one of the few “silver bullet” candidates for delivering successful software projects (F. P. Brooks, 1987). A survey by Larman and Basili catalogs many more refinements of the iterative software process, covering iterative practices from the 1930s to the 2000s (Larman & Basili,
Larman’s discussion also covers many of the various reasons behind the adoption and subsequent failure of the waterfall model.¹

Test-driven development also contributed to the agile methodology. In the iterative work discussed above, the various proposals included verifying the behavior of the program after each iteration. Each working program would be tested against the specifications implemented so far. Larman and Basili discussed some early examples of this testing turning into test-driven development (where the tests were written first) as early as NASA’s Project Mercury in the 1960s.

Other practices were also found to be helpful. Code review, customer participation, and frequent integration were found to be helpful. Beck assembled a number of these methods and created a framework where these methods worked together to create an agile software development process, which he called extreme programming (XP) (Beck & Andres, 2004). The “extreme” name was derived from the idea that a good idea, taken to extremes, might be better. If frequent testing was good, constant testing might be better. If periodic software integration was good, constant integration might be better. If periodic code review was good, constant code review (i.e. “pair programming”) might be better. Beck’s work, and that of collaborators, established a number of principles and practices of extreme programming that included test-driven development, continuous integration, and very short release cycles. Eventually those principles and

¹ This paper is truly an excellent discussion of the origins of modern software practice. The detailed history is a pleasure to read. It is readily available on the Internet in PDF form at http://www.craiglarman.com/wiki/downloads/misc/history-of-iterative-larman-and-basili-ieee-computer.pdf.² There is some code coverage support in GCC, but we wanted to implement
practices became known as the “Agile Manifesto”, summarizing the choices, values, and practices of the agile methodology (Fowler & Highsmith, 2001).

Beck made no claim to the uniqueness of any specific feature of XP, remarking that many parts of XP resembled how programmers acted “in the wild”, making it a good fit for a programming team interested in a more natural process.

A number of other agile frameworks have created variations on this theme, emphasizing various aspects of agile methods. SCRUM implements more management details than traditional XP (Schwaber, 1995). The Crystal methodology and Rational Unified Process (RUP) implement other aspects of team interaction. Williams does an excellent job of summarizing these and other variations in a recent survey of agile methods (Williams, 2010).

### 2.4.3 Agile Practices

For our purposes, we consider a process agile if it incorporates three basic practices for enabling software development in an environment of change. These practices are

- Development based on addition of small, well defined features with defined periods of time for implementing these features.
- Test driven development, where tests are written to specify and completely test each feature.
- Continuous integration of code as it is checked into a code repository, including running all past tests.
Small, Well-Defined Features

Agile development views the software as a collection of small, well-defined features. Each feature should be able to be completely understood and coded within a short period of time, typically a few days. The feature should be complete and testable, even if no other feature that follows it is implemented in the software.

Larger capabilities of the program are to be understood as a collection of features, which, if all implemented, would provide the larger capability. It is generally understood that if a subset of the features were provided, then ideally a subset of the capability would be available. However, there may be cases where some of the features in the collection are hidden or disabled until the entire collection is available.

Typically, the development of a feature is assigned to one or two developers who do nothing else until that small feature is completed and tested via an automated test.

Features are developed during specific time periods. Features are not worked on in advance of this time period. However, after this time period, the feature may be refactored – implemented differently – as long as the tests that define the feature continue to pass.

Test Driven Development

In test driven development (TDD), software development is seen as the controlled addition of a series of features to the evolving software product (Beck & Andres, 2004). To add a feature, first an automated test is written as a software function that exercises that feature in the software. Successful completion of the test is considered to be evidence
that the feature has been correctly implemented in the software. In this sense, the feature is exactly and only the behavior that the function tests for.

After the test is written, it is executed against the existing software system and observed to fail. The failure is expected, as the feature is not yet implemented. Then the developers add code to the program under development, with the goal being to pass the test. When it passes, the development is done. This process is shown in Figure 2.1.

Figure 2.1 The test driven development (TDD) workflow

This seems counterintuitive – that simply passing the test should be sufficient. But the we may observe that if a poorly implemented feature passes the test, then the test
should be made more discriminating, until a poor solution fails and a good solution passes.

One principle of TDD is that the simplest workable solution is sufficient, and that no preparation is made for features to be implemented in the future that are not part of the current feature set. To allow for the code to be safely modified when new features are added, the tests that validate the feature are retained for the duration of the software product lifetime. The collection of all tests constitutes a large collection of automated regression tests.

TDD has been associated with improvements in delivered quality. Williams reported defect reductions of 40% in teams at IBM following test-driven practices over teams using more traditional approaches (Williams, Maximilien, & Vouk, 2003). Further studies with teams at IBM and Microsoft were found to provide defect density decreases of between 40% and 90% relative to teams not using the practice (Nagappan, Maximilien, Bhat, & Williams, 2008). Saff and Ernst tested performance of students using automated testing tools. They found that for students using TDD tools, the time spent on programming problem sets was not significantly reduced, but that students who used TDD tools were several times more likely to complete the assignment correctly compared to students not using a tool, and students using continuous testing were twice as likely to succeed compared to occasional testing (Saff & Ernst, 2004).

It is important to remember that in agile processes, tests are much more than feature confirmation tools. Tests are feature definition tools. Agile features in TDD are
defined by tests that pass rather than any specific implementation or external description of the feature.

It is precisely this interpretation of the nature of features that makes the obvious solution to feature location – display the code that was changed to initially create the feature – unreliable. As the code is modified and enhanced over time, the feature implementation may be combined with other features, moved, delegated, or otherwise modified. As these refactoring activities take place in complex code bases, it is not always obvious what features are being affected by these modifications.

It is only the continuing passing status of the feature-verifying code that asserts that the feature is still present and capable, regardless of its new (and potentially unknown) location.

Continuous Integration

When software is written by a number of developers, the concurrent activity means that various changes are being made to different areas of the code simultaneously. Sometimes these changes may be conflicting.

In traditional software development, integration of various modules happens at periodic intervals, when all modules are compiled together and conflicts are detected, either via compilation errors or testing results. In a continuous integration environment, this integration is done continuously via automated systems (Fowler, 2006). Whenever new code is checked into the source repository, the automated system retrieves a complete copy of the code for the software product, builds it in its entirety, and executes all unit tests accumulated since the beginning of the project. A failure of any test requires
that the reason for the failure be determined, and corrective code is checked in before software development continues. Continuous integration effectively reduces the time between introduction and detection of defects, resulting in higher overall code quality and less time lost to late-phase defect discovery (Duvall, Matyas, & Glover, 2007).
CHAPTER 3

A Model for Extraction of Agile Features

Before exploring the specifics of the proposed research experiment, we needed a model of the agile process and the artifacts produced by this process in order to be able to determine the relationships and usage of the various data we find in the process. In this discussion we are using “agile artifacts” to mean the data produced by the process, and not the data used to model the software being created by the process.

In this chapter, we review existing models, define the principal artifacts of the agile process that are relevant to feature location, and then observe some important relationships we used in selecting feature-invoking and feature-excluding tests for feature location.

3.1 Agile Artifacts

As the processes of agile development are carried out, they produce a number of artifacts that contain information about the state of the software development process at various points in time. These artifacts have known properties that can be utilized to help make determinations about the software progress, testing state, code body, and other various aspects of the development effort.

3.1.1 Feature Descriptions

Agile processes model development as a sequence of feature additions, and each one of these features is initially conceived and documented as a textual feature
description. In automated systems, these feature descriptions, or “story cards” in XP vocabulary, are stored in a database along with details about which iteration, or development cycle, in which they were started and finished.

3.1.2 Iteration Plans

An automated agile process has a database of events for each iteration, or development cycle, that indicates which features were started in this iteration, which were finished, and which were not completed. This may be integrated with the feature description database, above, or kept separately in project management software.

3.1.3 Source Code Repositories

Agile processes relying on continuous build and integration practices to insure code stability need a central location to combine individual coding efforts. This is typically a source code repository, where changes to source code can be placed into the repository and tracked by an automated system. The code repository tracks, at a minimum, each change to a source file that is checked in, the time of the check-in, and the identity of the user. Some repositories may track the intent of the check-in, associating the change with a particular activity or feature, but most of the repositories currently in use in industry offer poor support for this. Repositories may also be able to manage multiple conflicting changes, and manage a merging process, but this is also not well supported in popular commercial systems such as Microsoft’s Visual Source Safe and Serena’s PVCS product (Atwood, 2006; De Smet, 2009; Pearce, 2005).
All major source control systems are able to retrieve a file, or collection of files, as the files appeared at a specific point in time, or, if a label was applied, at the time of the labeling action. (A broad summary of features is available online (Wikipedia, 2012).) Retrieval of files by time-based tags (or tags indicating another event tied to a build or iteration) is a critical characteristic, as it allows recovery of the code in various stages of development.

Continuous Build Records

Each time a continuous build and integration activity takes place, an automated agile system will make a record of that build and store those details for future examination. The build record will typically include a list of the files contained in the build, a label or time information allowing those files to be retrieved from the source control system, a list of tests that were run on the resulting software product, and the pass/fail record of those tests.

Depending on the product, these records may be kept in log files, in databases, or in HTML files used to present build history to developers.

3.2 Previous Models

There are not a lot of efforts involved in developing rigorous standard models and nomenclatures for agile processes. Part of this is caused by the fragmentation of the word “agile” to mean, basically, “capable”, in the marketing of competing software methodologies. There are models of specific processes that are referred to as agile – for instance there are references cited above for the SCRUM process, the CMMI agile
process, and the Rational Unified Process. These models are dissimilar to the extent that their processes differ, but they all contain some aspects that define iterations, testing, completion, and continuous integration practices.

Since we are only concerned with the aspects of agile methodologies required to implement feature location, we will draw a common subset of these models to create a simple description of the required abstract artifacts and relationships in agile methods.

3.3 Definitions

Let a software program $P$ progress through intermediate programs $P_0, P_1...P_n$ as it is modified over time from an empty program to the current program.

Let $C$ represent the series of changes $C_0...C_n$ where change $C_j$ is the change that changed program $P_{j-1}$ into program $P_j$.

Let $F$ represent the series of features or desired behaviors $F_0...F_n$ initially implemented by these changes, where $F_j$ is the feature implemented by $C_j$.

Let $T$ represent the series of test suites $T_0...T_n$ that test (i.e. verify the correct presence of) the features $F_0...F_n$.

A change set $S_k = [C_k, F_k, T_k]$ consists of the change $C_k$, the implemented feature $F_k$, and the verifying test suite $T_k$.

Finally, let $D$ represent the series of dates $D_0...D_m$ dividing the time periods (i.e. iterations or sprints) during which changes are made to add and test features in the program.
3.4  Modeling Agile Practices

As discussed above, we consider a process agile if it incorporates three basic practices for enabling software development in an environment of change. Here we look at the relevant practice rules of agile processes and restate them in terms of the model.

1. Development is based on addition of small, well defined features with defined periods of time for implementing these features.

   For some time period $D_j \ldots D_{j+1}$, let the changes, features, and change sets completed during this time be called $S_p \ldots S_r$. For all $q$ in $p \ldots r$:
   
   • On date $D_j$, no evidence of change set $S_q$ is in the repository.
   • On date $D_{j+1}$, change set $S_q$ is completed, tested and in the repository.

2. Agile processes use test driven development, where tests are written to specify and completely test each feature.

   At time $D_{j+1}$, for $q$ in $p \ldots r$, and for change set $S_q = \{C_q, T_q, F_q\}$, test $T_q$ completely exercises feature $F_q$.

   While the initial change $C_q$ that implemented $F_q$ may not remain as the implementation code for that feature, $T_q$ will continue to completely exercise that feature regardless of implementation location. In other words, after refactoring, $T_q$ will continue to exercise $F_q$, but may or may not continue to execute the code changed by $C_q$.

3. Continuous integration of code as it is checked into a code repository, including running all past tests.

   When $C_q$ is introduced, resulting in $P_q$, $T_q$ exercises $F_q$. The continuous build rule means that $T_q$ exercises $F_q$ for all $P_x$, $x \geq q$. 
3.5 Identifying the Invoking Test

Rule 1 states that at any $D_x$, all existing features are complete. Rule 2 states that $T_q$ exercises $F_q$ completely. Rule 3 states that this is true for all $P_x, x>q$. Therefore $T_q$ can serve as an invoking test.

If the program has been refactored (i.e. reorganized so as to execute some features differently), $T_q$ will still execute $F_q$ as long as $T_q$ is the operational definition of the feature. If there are new features added that replace or augment $F_q$, then obviously $T_q$ will not cover these new features, but there will be other tests that will. $T_q$ will continue to test $F_q$ to the extent that it exists in the code. If $F_q$ no longer exists, then $T_q$ will not pass and will have been retired, since by rule 3, all active tests must pass in a build.

If $F_q$ is modified in such a way that the feature is delivered differently, then by Rule 2, test $T_q$ will be modified accordingly to create the current definition of the feature. It is important to note that the invoking test is always the latest version (from a version-control point of view) of the test $T_q$, if $P$ as tested is the latest version, or else the version of $T_q$ is contemporaneous with the time of the version of $P$ being tested.

3.6 Identifying the Excluding Test

The purpose of the excluding test is to create a coverage set which, when subtracted from the invoking test coverage, leaves the coverage related to the feature of interest.

It is important to note that to accomplish this, we don’t need a single explicit test, but rather a method to produce the excluding coverage set.
Let K be a coverage set consisting of all the lines of code from P that are executed in a particular scenario or test. Specifically, $K_q$ is the set of lines of code, the coverage set, generated from running test $T_q$ exercising feature $F_q$. $K_q$ is the coverage set for $T_q$.

Note that coverage sets can be combined, where $K_{i...j}$ consists of the union of sets $K_i...K_j$. In other words, any line executed by any test $T_i...T_j$ is included in $K_{i...j}$.

Recall that at date $D_j$, change sets $S_{p...r}$ do not yet exist. Let $P_{p-1}$ represent the program state at date $D_j$. Since by rule 1, all features in $P_{p-1}$ are complete, no test $T_0...T_{p-1}$ can invoke any feature in $S_{p...r}$.

The program $P_{p-1}$ consists of $P_0$ after the application of change sets $S_0...S_{p-1}$. Each of these change sets implements a feature $F_k$ with test $T_k$. By rule 2, each $T_k$ completely tests $F_k$. The tests $T_0...T_{p-1}$ completely test all features in $F_k$ in $P_{p-1}$. The coverage set $K_{0...p-1}$ completely covers all features $F_0...F_{p-1}$.

Since the coverage set $K_{0...p-1}$ completely covers all features preceding $F_p$, it serves as a reasonable choice for a masking test for excluding features in $F_q$, $p <= q <= r$.

Another way to look at this is to consider $K_p$ as a test executing many features including the feature of interest. Ideally, the execution of any feature other than the feature of interest would execute code already tested by a previous feature test. Hence the union of all previous tests serves as a mask for the feature of interest.

One weakness in this argument is that $K_{0...p-1}$ do not serve to mask features created in the same iteration, i.e. in $F_p...F_r$ from $F_q$. If the test $T_q$ uses $F_x$ ($p <= x <= r, x != q$) then $K_{0...p-1}$ will not serve to exclude that feature. It is possible that features within an iteration (i.e. in the set $F_p...F_r$) are sometimes interrelated and thus not separable by this method. It
is not clear if this is a common occurrence, or if this issue is significant in the practical application of this method. It is certainly possible that in a feature location task, narrowing to one or two features out of a large body of code may still be helpful. This is one of the weaknesses we evaluated in the experimental portion of the study.

It is also possible that previous feature tests do not completely exercise the feature as they are intended to do, so that current use of an old feature exercises code not masked by previous tests. Again, this was a subject for experimental evaluation, to determine if such coverage gaps are widespread and if they significantly detract from the usefulness of the method.

3.7 Relating the Model to Agile Artifacts

A reasonable model should be able to find its components in the real world. In this case, we want to be able to identify the location of key items within agile artifacts.

- **Dates (D)**: These are the dates of the starts and ends of development cycles, possibly called iterations or sprints, and will be found in whatever iteration management software is being used.

- **Features (F)**: For each time period, or iteration, of development, a certain number of features will be implemented. The descriptions of these features are in the iteration story cards, which are short descriptions of required functionality. These are usually maintained in a simple database or spreadsheet.

- **Changes (C)**: Source code changes are maintained in a source control system. It is not necessary for this method to have changes to source code
accurately tagged with intended feature, but it is necessary to be able to retrieve changes in test code by date.

- **Tests (T_i):** Since tests are implemented with automated tools, tests are also stored in the source control repository. It is important to be able to retrieve tests by date and specific tests by feature; in other words for Fq, we need to be able to find Tq. This is generally done via naming and directory organization.

- **Programs (P_i):** Since agile projects build continuously, the binary program gets built frequently. Most of these builds are only diagnostic, and the binary is not retained for release. When the binary is retained, it is copied from the build machine to a packaging or retention mechanism. The details of this are not relevant to the feature location mechanism.

- **Coverage Sets (K_i):** In many continuous build systems, code coverage is computed for purposes of evaluating test completeness, using code coverage instrumentation tools provided by the software vendor or language author. Coverage data are rarely retained. For purposes of this method, the coverage set must be represented in a reasonable way and maintained, indexed by feature, in a way that allows rapid retrieval and combination.

### 3.8 Identification of Containing Tests

In an agile build process, all previous feature tests are executed and must pass before a build is accepted as complete. If a test fails, that feature is broken and must be
repaired. To do this effectively, each test must have some label, tag, or other property indicating which feature has just failed to execute properly. (Obviously there are also compiler messages or run-time errors that may also be helpful.) This set of properties can be used to map tests onto features, where the comprehensive list of features is found in the iteration data described above.

In practice, a method for naming tests is frequently used where test suites name the unit being tested, and individual tests are each named after a feature or aspect of a feature being tested (Beck, 2008). For instance, a test suite might be named SalesTaxManagerTest, while a test in that unit might be testSalesTaxExemptionForFood. Simply concatenating suite name and test name gives a good identity label for relating the test to the feature.

Identifying the invoking test (or tests) is therefore a matter of scanning the feature list and using whatever test property mapping exists to locate the tests for that feature. As a practical matter, software developed for this purpose would abstract these transactions so that the details of these lists and mappings for a particular software system are isolated from the logic of identifying the tests.

After identifying the invoking test, the test is executed within the software system, and the code coverage set collected. Again, a software abstraction should allow the storage of coverage sets by feature. Rapid storage and retrieval of these sets is essential to effective use of this method.

As a practical matter, it is not expected that these coverage sets would be created on demand, but rather created in the course of a conventional build and test event, where
all tests are run (as is the normal case in agile builds) and coverage sets are retained keyed to the features they test.

3.9 Creation of Excluding Coverage Sets

Since the running of tests is time consuming, it would be much more effective to compute excluding coverage sets in advance for each feature. This involves creating a partial ordering for the tests, so that all of the tests created in each particular iteration are known. As we execute the tests, we accumulate the union of the resulting coverage sets in a code coverage “accumulator” set. The process is outlined in Figure 3.1.

```
Initialize a code coverage set (the accumulator) to empty.
For each iteration
    For all feature tests completed in this iteration
        Run the feature tests to create the feature’s inclusion coverage set
        Copy the accumulator to serve as the feature’s exclusion coverage set
        Mask the inclusion set with the exclusion set
        Save this with test id to create the feature location set
    For all tests completed in this iteration
        Execute the test
        Collect the coverage set for the test.
        Set accumulator to union(current accumulator, test coverage set)
```

**Figure 3.1. Sequence for processing code coverage in iterations**

This process is executed in advance. Unlike the continuous build requirement of simply executing all tests, here we are imposing the partial ordering of executing tests in the order of their iterations. At the end of this process, we have exclusion coverage sets (i.e. the substitutes for the excluding tests) for all features.
3.10 Risks and Concerns

There are several concerns about the applicability of this method that might reduce its effectiveness. Part of the purpose of this study is to assess the degree to which these concerns affect the usefulness of the method.

- **Intra-iteration blur** – If two features interact and are both developed in the same iteration, mutual dependencies may make it difficult to claim that tests from before the iteration will adequately mask the feature.

- **Feature-selection boundary code** – It is possible that code exists to select a branching path depending on whether the feature is in use or not, or to decide how many times to execute a feature. Since this code is always executed, it won’t be detected as part of the feature by a masking approach. It is not known if this is a common problem, but it has been observed in prototype investigation code.

- **Confusion** – It is not known if the additional information provided by feature extraction will be clear enough to the user to be of practical use in feature extraction. Part of this concerns the design of the tool, but part of it is due to the nature of the information being presented and the ability to communicate that information in a clear and reliable model to the user of the tool.

- **Refactoring** – Agile software is continuously refactored, so that the location of feature code changes from time to time. If a feature is refactored in such a way as to be re-implemented via re-use of a later base
feature, masking code from before that iteration won’t mask the later base feature, so that code will show up as part of the selected feature.

Part of the study is to conduct experimentation with real world projects and professional programmers to determine the extent of the effect these issues have on the utility of this method.
CHAPTER 4

Pilot Investigations

Initial investigations of coverage comparison were done with small projects – e.g. expression parsers – written in ANSI C, Python and C# and using the repositories described above. Various features were implemented in simulated iteration periods. Prototype systems were created to collect and combine coverage sets and manually extract features based on combining and subtracting those sets. The feature identification worked as expected, though these are very small programs – a few hundred lines at most.

4.1 Prototyping Data Collection Systems

In order to collect the necessary data for feature extraction, we need to be able to collect code coverage in a scenario, and store that code coverage trace along with the information necessary to retrieve it by test and iteration.

Recall that in Section 2.1.2 we discussed Edward’s four conceptual components of an instrumentation system – insertion, runtime, collection, and analysis engines. We have implemented code coverage systems using this pattern for ANSI C, Python, and C# using three very different methods, each sufficient to serve our research purposes.
4.1.1 Code Coverage in ANSI C Using Code Rewrite

To instrument ANSI C in a portable way, we decided to approach the problem as the source-code level, since there is no native support in ANSI C for coverage collection\(^2\). We did this in a multistep approach.

First, we converted the C to an equivalent, reversible XML representation using srcML (J.I. Maletic, 2004). The srcML tool has been found to be useful in code transformation efforts (Collard, Decker, & Maletic, 2011). We planned to use code transformations to instrument the C code by injecting calls to coverage recording code.

Our original srcML representation contained the original code as embedded data, with every semantically significant structure in the code identified by XML tags. We then used an Awk script to scan the source file and add line numbers as attributes to all tags that corresponded to active lines of code, changing tags like

\[
<function>
\]

to tags like

\[
<function line="13">
\]

Next, we used an XSLT transform as the insertion engine to use the line number attributes to inject additional code of the form

\[
__LINE__ (line_number)
\]

in front of each code item that caused a line execution (where \textit{line\_number} is the actual recorded line number), and injected brackets as necessary to maintain semantic

\[
__________________________
\]

\(^2\) There is some code coverage support in GCC, but we wanted to implement coverage in a portable way, and we needed to be able to add hooks for tagging with test information.
equivalence. This also included the XML tags necessary to support the reverse srcML transformation.

We also added an inclusion of a “coverage.h” file at the beginning of each source file that defined the \_LINE\_\(n\) insertion as a runtime macro that recorded the execution of line N.

Finally, we ran the reverse srcML transformation, removing the XML tags, leaving just the instrumented code. When this code is executed, coverage data are collected and saved to a file. The utility package that is running the tests then needs to get the coverage from the file, associate the data with feature and test information, and store the data into a data store.

The Awk and XSLT files for this method, and sample source files before and after instrumentation, are included in Appendix Section A.1.

4.1.2 Code Coverage in Python Using Trace Interception

To collect coverage information from Python, we implemented a small utility library that uses Python’s built-in settrace function to implement coverage capabilities. The settrace function can be used to specify a function to be called after every Python statement. It serves as a built-in insertion engine. The trace function is called with frame information including the file and line being executed. It is a simple matter to capture that information at runtime and retain it along with information about the test being executed at the time.

The settrace function that we implemented was straightforward, saving each line execution event in a Python dictionary:
def _trace(frame,x,y):
    if (frame.f_code.co_filename[1] == 'U'):
        _data[(frame.f_code.co_filename, frame.f_lineno, _story, _number)] = 1
    # print frame.f_code.co_filename
    # print frame.f_lineno
    return _trace

Figure 4.1. The Python tracing function used with settrace

Note that the _story and _number global variables (which are set before each test) are recorded along with the coverage information. For data collection at the end of the run, the _data dictionary is then saved to record the tagged coverage information.

To record the associated test information for each test, the python pyunit test framework was used to organize the unit tests. We extended the test framework class to save the coverage tagging information so that the trace function could capture it in the recording process.

The Python coverage.py file is including in Section A.2 in the appendices, including the calc.py file being tested, and testcalc.py containing the unit tests.

4.2 Simulating Iterative Development and Feature Extraction

We created a small software development effort to simulate adding features to code in an iterative environment. Our purpose was to create a small instance of the agile artifacts described in Chapter 3, and then to use those to extract features according to the proposed method.

The test software created is a small software library (calc.py) designed to evaluate simple arithmetic expressions presented in strings. An example call to the library looks like this:
Value = Calc.Evaluate("1+4.3*5.2")

We coded the library in three iterations, with two feature stories in each iteration.

The iterations were as follows:

**Table 4.1. Simulated iterations**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Feature/Story</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processes single-digit integers</td>
<td>SD</td>
</tr>
<tr>
<td>1</td>
<td>… positive integers</td>
<td>PI</td>
</tr>
<tr>
<td>2</td>
<td>… negative integers</td>
<td>NI</td>
</tr>
<tr>
<td>2</td>
<td>… floating point numbers</td>
<td>FP</td>
</tr>
<tr>
<td>3</td>
<td>… add/subtract expressions</td>
<td>AS</td>
</tr>
<tr>
<td>3</td>
<td>… multiply/divide expressions</td>
<td>MD</td>
</tr>
</tbody>
</table>

For each iteration, feature/story tests were added to the test collection. Two tests were added for each iteration, with each test thoroughly exercising the feature. Each test followed the same pattern:
# Story 1: Single Digits

# -- evaluate should raise ValueError exception for non-strings
# -- evaluateDigit should raise ValueError exception
# for strings where length != 1
# -- evaluateDigit should raise ValueError exception
# for non-numeric strings
# -- evaluateDigit should return accurate single digits
# -- evaluate should return accurate single digits

class SingleDigitTestCase(coverage.TestCase):

    story = "Single Digits"
    story = "SD"
    number = 1

    def testNonStringRaisesException(self):
        self.assertRaises(ValueError, calc.evaluate, 1)
        self.assertRaises(ValueError, calc.evaluate, {""})
        self.assertRaises(ValueError, calc.evaluate, ["" ])

    def testBadLengthRaisesException(self):
        self.assertRaises(ValueError, calc.evaluateDigit, """)
        self.assertRaises(ValueError, calc.evaluateDigit, "00")
        self.assertRaises(ValueError, calc.evaluateDigit, "000")

    def testNonDigitRaisesException(self):
        self.assertRaises(ValueError, calc.evaluateDigit, " ")
        self.assertRaises(ValueError, calc.evaluateDigit, "-")
        self.assertRaises(ValueError, calc.evaluateDigit, "x")

    def testSingleDigitReturnsCorrectValue(self):
        self.assertEqual(calc.evaluateDigit("0"), 0)
        self.assertEqual(calc.evaluateDigit("9"), 9)

    def testEvaluateReturnsCorrectValue(self):
        self.assertEqual(calc.evaluate("0"), 0)
        self.assertEqual(calc.evaluate("9"), 9)

Figure 4.2. Example feature/story test code

After the tests were added to the test collection, code was written and added to the calc.py library to successfully execute the tests. When the tests were running successfully, a code coverage run was used to run the tests and collect the coverage for each test. This collection of coverage information was done at various points during the development process.

After collecting code coverage information, a simple method was used to display code coverage information. We modified a code listing by adding a comment to each line
containing the codes for each feature that exercised that line when tested. For instance, here are some commented statements:

```plaintext
48: while cursor < len(s) and s[cursor] in "*/": #SD #PI #AS #MD
49:   op = s[cursor] #ND
50:   cursor = cursor + 1 #ND
```

These indicate that line 48 was used for the SD, PI, AS, and MD features (i.e. some common code) but that lines 49 and 50 were only executed by the MD feature, indicating that 49 and 50 are unique to that feature. Only the first feature to appear after the “#” comment indicator would be left after masking, so only 49 and 50 would be part of the MD feature specifically.

These type of listings were examined to verify that the behavior predicted by the model was in fact observed. Results are discussed in the next section.

### 4.3 Pilot Feature Extraction Results

We tried feature extraction at three points: after the second iteration, after a refactoring necessary to prepare for the third iteration, and after the third iteration. We will discuss and offer some observations from each of these results.

#### 4.3.1 First Iteration

Code coverage after the first iteration would not have had masking tests, so would not be applicable to the proposed method. No testing was done at this point.
4.3.2 Second Iteration

Coverage results were collected after implementing the second iteration – negative integers (NI) and positive floats (PF). The results of the second iteration are presented in Figure 4.3.

Ran 10 tests in 0.007s

OK
1:  #calc.py
2:  import string
3:  
4:  def evaluateDigit(s): #SD #PI #NI #PF
5:    if len(s) != 1: #SD #PI #NI #PF
6:      raise ValueError #SD #PF
7:    return string.atoi(s) #SD #PI #NI #PF
8:  
9:  def evaluate(s): #SD #PI #NI #PF
10:    if (type(s) != type(“”)): #SD #PI #NI #PF
11:      raise ValueError #SD
12:    v = 0 #SD #PI #NI #PF
13:    multiplier = 1.0 #SD #PI #NI #PF
14:    cursor = 0 #SD #PI #NI #PF
15:    decimal = False #SD #PI #NI #PF
16:    
17:    #check for negative sign #SD #PI #NI #PF
18:    n = 1 #SD #PI #NI #PF
19:    if (s[cursor] == ‘-‘): #SD #PI #NI #PF
20:      n = -1 #NI
21:    cursor = cursor + 1 #NI
22:    
23:    #process digits #SD #PI #NI #PF
24:    while cursor < len(s): #SD #PI #NI #PF
25:      c = s[cursor] #SD #PI #NI #PF
26:      if c.isdigit(): #SD #PI #NI #PF
27:        if (decimal):
28:          multiplier = multiplier / 10.0 #PF
29:          v = v + evaluateDigit(c) * multiplier #PF
30:        else:
31:          v = v * 10 + evaluateDigit(c) #SD #PI #NI #PF
32:        else:
33:          if c == ’.’:
34:            if (decimal):
35:              raise ValueError #PF
36:            else:
37:              decimal = True #PF
38:        cursor = cursor + 1 #SD #PI #NI #PF
39:        return n * v; #SD #PI #NI #PF
40:        else:
41:          return n * v; #SD #PI #NI #PF
42:  
Figure 4.3. Source code with trace information after second iteration
There are several interesting observations to be made here. First, of course, is that the features from the second iteration (negative integers (NI) and positive floats (PF)) are detected in lines 25-26, and lines 33-38 in lines where those coverage traces are not masked by traces from earlier features. While the algorithm does not detect the complete feature, it certainly points out critical lines in those features’ implementations. This is consistent with Wilde’s observations that while software reconnaissance doesn’t always identify the complete feature, it is useful for drawing attention to the location of the feature implementation (N. Wilde, 1994).

Software reconnaissance is very good at finding unique code, that is, code that does nothing but implement one specific functionality. However it does not typically find all the code that the maintainer needs to study to understand that functionality… Software reconnaissance should be seen as a good method for finding places to start looking when dealing with strange code. (N. Wilde, 1994)

The second observation here is that the method does not detect what we will call the boundary code around a feature. Boundary code is the code that decides whether to invoke or not invoke, or how many times to invoke, feature code. The reason for this is that even excluding tests invoke the boundary code in deciding not to execute the feature. This is also consistent with Wilde’s observations. Additionally, this means that a feature existing entirely of boundary code (an extremely small one, for instance) might not get detected.

Third, we observe that the single digit (SD) feature coverage is essentially the same as the positive integer (PI) code. Though both of these are from the first iteration, we know that the SD implementation was substantially smaller than the PI code. Further reflection, however, indicates that once we created a multidigit integer capability, the
single-digit integer just became a one-iteration version of the general multi-digit case. In other words, an earlier feature was essentially reimplemented as a minimal case of a later feature. Something similar would happen, for instance, if a sales tax feature implemented “no sales tax” as “sales tax rate = 0.0” – an attempt to mask a sales tax test with a no-sales-tax code test would fail to provide useful results.

4.3.3 Preparatory Refactoring

At the beginning of the third iteration, it was necessary to do some refactoring of the calc.py library to prepare for an additional level of the recursive descent parser being designed. We wanted to know how durable the feature identification would be now that the features were implemented in different functions.

The listing from this refactored code, based on running the same tests, is shown in Figure 4.4.
Figure 4.4. Source code with trace information after refactoring

The primary observation to be noted in this listing is that the features (NI, PF) are located in exactly the same way despite having been moved from evaluate() to
evaluateNumber(). This is the behavior we are looking for – that feature identification can be made robust in the presence of source refactoring. Note that these were the same tests as used in the second iteration, so the definition of the features remained unchanged.

4.3.4 Third Iteration

In the final iteration of our simulated agile development effort, we implemented simple arithmetic expression evaluation. Simple addition / subtraction (AS) and multiplication / division (MD) expression processing features were added, including in the definition of those features such things as left-to-right processing and respect for operator precedence when multiply and divide were added.

The listing for this coverage result is somewhat longer. Note the very nice identification of the AS and MD features in lines 49-68 in the listing in Figure 4.5.
Ran 19 tests in 0.014s

OK

1: #calc.py
2:  import string
3:  
4: cursor = 0
5:  
6: def evaluateDigit(c):
7:   if len(c) != 1:
8:     raise ValueError
9:   if not c.isdigit():
10:      raise ValueError
11:   return string.atoi(c)
12:  
13: def evaluateNumber(s):
14:   global cursor
15:   if not s[cursor] in "-01234567890."
16:     raise ValueError
17:   v = 0
18:   multiplier = 1.0
19:   decimal = False
20:   #check for negative sign
21:   n = 1
22:   if (s[cursor] == '- '):
23:     n = -1
24:     cursor = cursor + 1
25:   #process digits
26:   while cursor < len(s) and s[cursor] in "01234567890."
27:     c = s[cursor]
28:     if c.isdigit():
29:       if (decimal):
30:         multiplier = multiplier / 10.0
31:         v = v + evaluateDigit(c) * multiplier
32:       else:
33:         v = v * 10 + evaluateDigit(c)
34:     elif c == '.':
35:       if (decimal):
36:         raise ValueError
37:     else:
38:       decimal = True
39:     cursor = cursor + 1
40:   return n * v;
41: 
42: def evaluateMulDivExpression(s):
43:   global cursor
44:   value = evaluateNumber(s)
45:   while cursor < len(s) and s[cursor] in "*/"
46:     op = s[cursor]
47:     if op == '*':
48:       value = value * evaluateNumber(s)
49:     if op == '/':
50:       #force float division
51:       value = (value * 1.0) / (evaluateNumber(s) * 1.0)
52:   return value
53: 
54: def evaluateAddSubExpression(s):
55:   global cursor
56:   value = evaluateMulDivExpression(s)
57:   while cursor < len(s) and s[cursor] in "+-"
58:     op = s[cursor]
59:     if op == '+':
60:       value = value + evaluateNumber(s)
61:     if op == '-':
62:       value = value - evaluateNumber(s)
Figure 4.5. Source code with trace information after third iteration

In this listing, as mentioned above, it is easy to see that the AS and MD features are being extracted reasonably well. Also note that in `evaluateAddSubExpression()` in lines 62-67, the lines are being identified as primarily implementing Add/Subtract. If, however, MD were tested first, it would have masked AD since it also uses that feature, had it been included in the masking set. This is why it is important that masking tests be used from previous iterations — they can’t generally call features that haven’t been implemented yet. (Of course this is also subject to the feature-reimplementation concern discussed above.)

We note that the feature extraction at this point in lines 49-55 and in lines 62-67 seem to be generally working well. However, lines 47 and 60 are not detected because they can’t be masked – they are now part of the implementation of earlier features like positive floats and single digits, serving as another example of the feature reimplementation concern. Lines 48 and 61 are not detected because they are the boundary code for the feature, both `while` loops that may execute many times, once, or in the case of excluding the feature, not at all.
4.4 Pilot Feature Extraction Conclusions

Our experiences with the pilot feature extraction project indicate that the weaknesses with the method do exist and are not uncommon, but do not interfere with the general ability of the method to locate the areas of implementation for specific features. The specific limitations we found were:

- Inability to detect feature code at boundary conditions
- Tendency of earlier features to expand their mask when reimplemented using later features, as for instance the large mask of the Single Digit (SD) code.
- Inability to detect features when non-use of the feature executes the same code as use of the feature, as in the “sales tax = 0” example.

It may be that these limitations are overly represented in the pilot study because of the small, densely coded nature of the example.

Even so, we think the results are promising from a practical point of view. While working on this pilot project, we showed these results to several professional programmers who indicated that the level of feature identification displayed by this system would indeed be sufficient to significantly assist them in feature location efforts.

Of course, this is preliminary work, and one of the goals of the proposed research is to determine if in fact the general ability of the method to locate features, despite these limitations of the method, is sufficiently powerful to contribute to the effectiveness of feature location tasks.
4.5 Designing the Feature Viewer Application

To conduct the experiment, we needed to implement a feature viewer that would allow users of the system to

- Browse the code under examination
- Search the text for features by keyword
- Browse the available tests that indicated feature location
- Search for relevant tests by keyword
- Indicate in the source code the lines selected by the text
- Allow the user to select lines of text considered to be in a feature
- Allow the user to save that selection to a named file

4.5.1 Existing Interfaces for Code Browsing

Many examples of code browsing tools exist, both in the literature (e.g. see various surveys (Simmons et al., 2006)) and in commercial systems. One paradigm that is very common is to display a directory hierarchy being browsed in one panel, and the text of one or more files of interest in another panel. Several examples are shown in Figure 4.6, Figure 4.7, and Figure 4.8.
Figure 4.6: Code browsing in OSX Code Runner
Figure 4.7: Code browsing in *Python Scripter*
Figure 4.8: Code browsing in *Visual Studio 2010*

Clearly this is a common presentation model and should be familiar to many professional programmers.

### 4.5.2 Prototyping the Feature Viewing Capability in the Viewer

As described above, we wanted implement a feature viewer with the ability to view a feature map (e.g. a processed coverage map) that was laid on top of the browsed code. This means that for any file or line of code, it should be possible to see if the coverage map or feature map currently being displayed, and to see which files and lines of code are included in the map. The basic paradigm for this is to indicate file inclusion in
the file directory tree, and line inclusion in the text browser. An example of this is shown in Figure 4.9.

![Figure 4.9: Prototype of code browsing by feature in viewer](image)

Second, we want to add the ability of the user to create a feature map either by using an automatically generated map as a starting point or by simply creating one from their own judgment while performing code examination. To do that, we need an independent way of indicating membership in the user feature map. An example of both maps being used simultaneously is shown in Figure 4.10, where the yellow bar indicates the user’s determination of the feature location.
4.5.3 Pilot Implementation of the Feature Viewer

The code viewer was implemented as designed in the prototype with the addition of a two major changes.
To assist in code browsing in large files, a small code browser map was implanted on the right-hand-side of the code-browsing window, and selection highlights were also shown in the map to make it easier to navigate a large file.

Due to limitations in the graphical toolkit, an alternate method of displaying the feature selection and user selection was implemented. The feature selection was shown in red, and the user selection was shown in blue; lines in both feature and user selections were shown in magenta. This gave a natural appearance to the selections and worked well in the main text and in the miniature code map.

The feature viewer was implemented in Python 2.7 using the Ttk toolkit and run during the experimental process on the OSX Mavericks operating system. The application as initially implemented is shown in Figure 4.11.
Figure 4.11: First working implementation of the feature viewer

In Figure 4.11, there is one test selected in the lower left corner, which indicates that it has highlighted six lines of code. In the files box above it, the program indicates that six lines are selected in the file “bottle.py”. In the code browsing window in the center of the screen, the six lines are highlighted in red. The code map at the right of the window also shows the six lines highlighted in red, in the context of a much larger view (i.e. “zoomed out”) view of the code.

While not shown in the figure, when the user selects code believed to be responsible for the implementation of a feature (i.e. the feature location) the selected code is shown highlighted in blue.
4.5.4 Pilot Experiment with the Feature Viewer

In a small pilot experiment, 8 participants, all professional software developers, were each asked to use the feature viewer to locate 10 features in the code. (Details of participant selection will be discussed in Chapter 5.) For each participant, half of the 10 features were located using the feature test capability to highlight feature locations, and half of the features were located using simple text search in the source code. The experiences of the participants were observed carefully during the 80 feature location tasks.

After these sessions, the participants took place in focus group discussions where they were asked what features made the program easy or difficult to use, and what suggestions they had for effectively presenting the feature-location interface.

Combining observations and discussions, the following issues were identified and modifications made to the feature viewer application.
Table 4.2. Issues identified in focus groups and resulting modifications

<table>
<thead>
<tr>
<th>ISSUE</th>
<th>MODIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>It was difficult to search for keywords in the feature test window, as no search capability was provided for searching names of tests or test suites.</td>
<td>Implemented text searching in the feature test window, as described below.</td>
</tr>
<tr>
<td>Having found a test that seemed related, users shifted focus to the source code without investigating other, more useful tests later in the list.</td>
<td>Visually highlighted tests that were found in the keyword search of feature test names, and removed other tests from view.</td>
</tr>
<tr>
<td>The text search feature was implemented by highlighting all locations of the text, which was felt to be less useful than a find-first/find-next paradigm.</td>
<td>A standard first/next paradigm was implemented, copying the Visual Studio search capability.</td>
</tr>
<tr>
<td>Feature location tests that did not identify lines at all (because all lines were masked) were not useful in the feature test list.</td>
<td>Feature tests that did not identify any lines were removed from the feature list. This decision is discussed later in chapter X.</td>
</tr>
<tr>
<td>The miniature code map was a distraction because most of the developers had never seen such a thing.</td>
<td>The miniature code map was removed.</td>
</tr>
<tr>
<td>Since the feature search was confined to one file, the file directory window was not useful.</td>
<td>The file directory window was removed.</td>
</tr>
</tbody>
</table>

The resulting modifications made the feature location tool much easier to use by removing user interface distraction that detracted from the ability of participants to understand and use the tool.

The data collected (time and confidence) were recorded and used for initial statistical evaluation and to allow coding and testing of the data analysis tool set. Some results of pilot investigation are discussed briefly in Section 4.6, later in this chapter, though they do not contribute to the final experimental results.
4.5.5 Final Implementation of the Feature Viewer

Incorporating the modifications described above, the final implementation of the feature viewer was created in Python 2.7 using Ttk, and executed during the experiment on the OSX Mavericks operating system. The feature viewer is shown in Figure 4.12.

The following specific changes were made:

1. The file directory listing and the miniature code map were removed.

2. A feature search bar was added to the top of the feature test list. In this view, the words “cookie encode” were searched for in the test list, which restricted the list to the test suite and the two contained tests that are related to this search. In the code windows, the six lines selected by the feature tests are highlighted, most of them in red, but one line (the current user selection) in magenta.
3. The direct text search bar, as can be seen, moved to the bottom of the code window, along with the up and down arrows allowing rapid movement to the previous or next instance of the searched-for text.

Feature Test Searching

4. The search algorithm in the feature testing tool was changed to allow keywords to be entered into the search box, and to only display tests where the keywords were all found in the complete test name (<test_suite> + <test_name>). This behavior is shown in Figure 4.13.
5. The search was updated so that as additional keywords were added to the search filter, the search narrowed to those tests containing all keywords. Figure 4.14 shows this behavior. Notice that several tests in this picture don’t contain the word “route” in their names, but the word is contained in their suite names. Again, both suite name and test name were searched.
Finally, we allowed excluding keywords to be added by prepending a minus ("-") sign to a keyword, causing tests containing that keyword in their full names to be excluded from the list. In Figure 4.15, the excluding keyword “-error” resulted in the removal of tests containing that word.
Figure 4.15: Adding exclusion keywords to the feature test list

At first, icons alone were used to indicate which tests had been found, but participants in the focus group felt that hiding the tests that did not match the search terms emphasized the useful tests more strongly and provided a better experience.

Source Code Text Searching

In implementing the source code text search, it was important to implement a search experience similar to ones found in modern development environments. The Visual Studio search model was adopted, which included:

- Case-insensitive searches
- Find previous / find next capabilities
- No capability for multiple keyword searches
- No capability for exclusion searches
- Regular expression search capability
We implemented this model, except we did not include regular expression capability, because 1) most IDEs do not include this, and 2) the focus groups indicated that they rarely used that feature and would be unlikely to do so while searching for features.

The resulting text search capability can be seen in Figure 4.16. The program presented dialogs indicating the number of times the text was found, and dialogs at each end of the search indicating that there are no more instances above or below the current location. The search method was accepted as routine without objection or discussion by the developers during the remainder of the study.

![Figure 4.16: Basic text searching in the source code window](image)
Saving Feature Locations

During each feature location task, the program allowed the feature test list to be hidden, or not, depending on the feature location mode used for that task. This was just to minimize distraction and emphasize the current mode to the participant.

After each feature was located by the participant, a code was provided to the participant to use in saving the location. These files were later collected and used to associate feature location lines identified with specific feature location tasks. This interface is shown in Figure 4.17.

![Save Location File... interface](image)

**Figure 4.17: Saving the feature location file.**

At this point we had a complete feature viewer, where a user could load a set of tests and coverages, view those coverages within the context of the code being explored,
use those coverages to help identify features locations, identify feature locations manually if necessary, and save location data to a file identified with the code for a particular feature location task.

Availability of the code for this application and related files is discussed in Section A.3 in the appendices.

4.5.6 Participant Behavior Observations

We verified that the application was quickly learned by participants who were familiar with the common design of modern software development environments. When pilot participants were asked if the task and the task environment seemed realistic, answers were positive. (Mean = 8.6 on a scale of 0 - 10, n = 8, std. dev. = 0.7)

While observing participants, we also noted that occasionally feature location tasks were interrupted by general lack of knowledge of the feature being examined. This happened in two circumstances.

First, some of the initial participants were not familiar with the general purpose of the program being examined (a web server) so did not understand the meaning of the feature being requested. For instance, asking about cookie encryption was more difficult if someone didn’t know what web cookies were used for at all.

Second, some of the initially selected features required deep knowledge of the internals of the particular web server being examined, which some developers found problematic. For instance, find the STPL parser isn’t a question about web servers in general, but a question about an implementation detail.
To address the first issue, we decided to be more specific about participant selection, limiting participants to software developers who both program professionally and have a general idea of the mechanics of serving web pages.

For the second issue, when determining features to be located in the experiment, we limited scope to features where a developer would understand the purpose of the feature before examining the program. We felt that this better matched the real-world scenarios where this kind of feature searching in an unfamiliar program would be taking place.

In other words, we might ask for a feature like “find where a request header date is evaluated,” but not a feature like “find where STPL processes a gen_template object”. While we felt that the method is applicable in this instance, we didn’t feel that the latter was typical for an initial feature location task, and is out of scope for this study.

4.6 Pilot Location Usability Study Results

While the statistical analysis of the pilot usability study results had little effect on the design of the feature browsing software, it did impact the design of the experimental protocol.

An independent-samples t-test was conducted to compare time-to-locate and confidence-in-location in text-search (control) and feature-search (experimental) conditions.

There was a slightly significant difference found in the time-to-locate scores for text-search (M=144, SD=125) and feature-search (M=111, SD=116) conditions;
t(80)=1.324, p=0.22. These results suggest at p=0.22 that feature-search lowers time-to-locate.

There was a slightly significant difference found in the confidence-in-location scores for text-search (M=6.4, SD=2.2) and feature-search (M=7.0, SD=2.1) conditions; t(80)=-1.414, p=0.16. These results suggest at p=0.16 that feature-search raises confidence-in-location.

It was observed by looking at individual scores that both individual performance and feature complexity appeared to vary widely. The data had not been collected in a paired fashion; location tasks were randomly assigned to control or experimental groups before execution. Because of this, it was not possible to conduct paired t-tests, which would have removed noise due to the effects of feature complexity and user performance variations.

To address this, we computed the overall mean time to locate each feature for all users, and the overall mean time to for each user to locate all features. Using this, we normalized feature completions times, separately for time-by-user and time-by-feature. These values were expressed as percentages of average times-to-locate. We then examined those results for experimental and control groups.

Another independent-samples t-test was conducted to compare time-to-locate-by-user and time-to-locate-by-feature in text-search (control) and feature-search (experimental) conditions.

There was a difference found in the normalized time-to-locate-by-user for text-search (M=115%, SD=85%) and feature-search (M=85%, SD=73) conditions;
t(80)=1.691, p=0.09. These results suggest at p=0.09 that feature-search lowers normalized time-to-locate-by-user.

There was a difference found in the normalized time-to-locate-by-feature for text-search (M=124%, SD=107%) and feature-search (M=77%, SD=61) conditions; t(80)=2.496, p=0.01. These results suggest at p=0.01 that feature-search lowers normalized time-to-locate-by-feature.

Clearly, the normalization benefit from removing the influence of task difficulty variation had produced a much clearer result, and even outperformed removing user-related variation.

Based on this, we concluded that removing the noise from feature variation was more effective than removing noise from user variation in examining the effect of the experimental method on time-to-locate performance. This suggested that removing the noise variables by incorporating a paired-test design would allow us to isolate the effect of the feature-search tool. This was the major change to the experimental protocol resulting from the pilot data analysis. In retrospect, it turned out to not be necessary as the signal from the benefit from the tool was much stronger than the noise from task or participant variation. This was especially true when we removed some participant variance by more carefully selecting participants as discussed in Section 5.4.

Details of this analysis are found in an IPython Notebook session file, available at the repository discussed in section A.3 in the appendices.
CHAPTER 5

Experimental Methodology

The purpose of the experimental portion of the proposed research was to assess the effectiveness of this method for programmers working in real-world agile environments and projects. To do this, we conducted a controlled experiment comparing the results of feature location tasks with and without the use of a tool implementing the described method.

Studies such as this, asking volunteers to execute programming tasks, are found commonly in the literature. Saff and Ernst used students to compare performance on programming tasks with and without various testing tools (Saff & Ernst, 2004). They relied on questionnaires and actual measured metrics to observe the effects those tools had on task completion. We modified this design to account for several aspects of qualitative analysis as discussed below, but used a similar experimental framework of timed performance and questionnaires.

5.1 Experimental Questions

The goal of the experiment was to assess the usefulness of the feature location method with respect to the performance of people using the method in feature location and identification tasks.
Given the source code for an application program, and a set of proposed feature locations from the test-based feature extraction methods, all displayed in a suitable tool, we had three quantitative measures of interest:

- Do software developers using the tool find features to their satisfaction (i.e. they believe they have completed the feature location task) faster than developers not using the tool?
- Do software developers using the tool feel more confident that they have correctly located the lines implementing the requested feature, compared to developers not using the tool?
- Do software developers using the tool more accurately identify feature locations compared to programmers not using the tool?

We would also like to know if developers feel, subjectively, that the tool is useful, and if they would use the tool in real-world situations.

5.2 Available Resources

We had several important software capabilities available at the beginning of this experiment.

- A code coverage tool for Python programs, instrumented to collect code coverage for individual tests under the python unittest framework.
- A code repository scanner that could gather agile artifacts such as commit dates, commit contents, and other data from a Git repository.
• A program that could process tests and artifacts according to the method described in Chapter 5, resulting in specific feature locations associated with many of the feature-testing unit tests.

• A program that allowed participants to view code being examined, search feature tests, display code identified as location content by feature tests, and record participant identification of lines involved in a feature location.

We also had some additional resources that were important.

• Permission from a large commercial software IT organization to recruit participants from within their staff of developers for research purposes, and for those participants to participate in these experimental activities during rearranged work hours.

• Permission to use conference rooms in the IT facility for the purpose of conducting experiments. (No other company-owned hardware or software was used in the experiment, except for using LCD monitors in the rooms.)

5.3 Preparatory Activities

Before actually conducting the experiment, some preparatory activities needed to be completed. Here we discuss identifying suitable code to search, extracting features, and the data processing necessary to create the feature location data necessary to use the test feature location tool in the experiment. All software used in this step of the preparation is available at the repository discussed in Section A.3 in the appendices.
5.3.1 Identification of Suitable Code to be Searched

It is often the case that academic investigations examine large open source projects because of the availability of the code and underlying repositories. We believed that in order to examine the research problem we needed to find a code for an application that had characteristics of commercial application software:

- Currently in use in commercial and industrial situations
- Developed by a group of people under real-world conditions
- Developed using an agile, test driven process
- Developed with artifacts typical of a commercial agile activity, including a complete record of commits of both application and test code.

After looking at a number of projects, we decided to use the code for the open source `bottle.py` web server available at http://bottlepy.org. This system is in use as the basis of many commercial web sites. It has been developed under agile, test driven practices with a complete set of commit records, which are readily available at https://github.com/defnull/bottle.

The `bottle.py` system application source is over three thousand lines of Python code, presented in one large file for ease of deployment. The system also includes several hundred unit tests, which were suitable for the feature location method described in Chapter 3.

5.3.2 Extraction of Agile Artifacts

In addition to downloading the source code for the `bottle.py` application, we also cloned the application’s Git repository to a local computer. We then created a verbose log
of the repository commit history, containing data for each commit including the date, purpose, and lines changed during that commit event.

We wrote a small utility in Python to parse this file. Since every line of source code had to be committed at some point, it was possible to find the first instance of every unit test. The date of first appearance of each unit test was used to assign that test to an iteration. (Any method of assigning tests to iterations would be acceptable. Since this project didn’t have formal iterations, we treated each commit, which involved a number of changes, as an agile iteration.)

5.3.3 Extraction of Unit Test Feature Locations

We used the modified unittest.py to run the unit tests included with the bottle.py application under coverage instrumentation, and saved the code coverage for each test into a data file. We then wrote a Python utility to read the coverage data and the test iteration data, and use the method of Section 3.9 to calculate the masked test coverage for each test. The masked coverage (i.e. the feature location for that test’s feature) for each test was saved in a data file, indexed by the fully qualified test name.

It is worth noting that all tools and methods used would have worked equally well had the feature coverage been spread across a number of files. Even in this case, coverages lines were recorded across multiple files, since other unit tests in their files were included in the coverage maps. All data were recorded with file information as necessary. For this particular application, only locations in bottle.py were relevant, so other coverage locations were discarded.

The feature extraction workflow is shown in Figure 5.1.
Figure 5.1: Feature extraction workflow.
Performance of the feature extraction method in extracting plausible feature sets, in terms of number of tests returning useful locations, size of locations, etc., will be discussed in the results section.

5.3.4 Identification of Features to be Located

In the focus groups, we identified major features of web servers, and created a question about each of seven features. These included finding locations for managing cookies, sending files, uploading files, encrypting cookies, authenticating headers, error handling, and expiration issues.

The exact terminology of the phrases used to describe the tasks is in Table 5.1.

<table>
<thead>
<tr>
<th>Code</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Locate the code that parses a date</td>
</tr>
<tr>
<td>B</td>
<td>Locate the code that saves an uploaded file</td>
</tr>
<tr>
<td>C</td>
<td>Locate the code that prevents sending an invalid file</td>
</tr>
<tr>
<td>D</td>
<td>Locate the code that encodes a cookie</td>
</tr>
<tr>
<td>E</td>
<td>Locate the code that authenticates a request</td>
</tr>
<tr>
<td>F</td>
<td>Locate the code that deletes a cookie</td>
</tr>
<tr>
<td>G</td>
<td>Locate the code that handles header expiration</td>
</tr>
</tbody>
</table>

5.3.5 Preparation of Experimental Workstation and Environment

To conduct the experiment, we loaded the feature viewing software discussed in Section 6.5.5 on a MacBook Pro laptop computer running the OSX Mavericks operating system. The data file containing the extracted feature location described above and the source code of the bottle.py application were loaded into the feature viewer program.
The computer was connected to a secondary monitor mirrored to the main display, and an auxiliary keyboard and mouse. This allowed a participant to sit in front of a monitor, keyboard, and mouse, and allowed the observer to monitor activities on the laptop screen.

The observer sat across the table, several feet from the participant, and observed the session on the mirrored laptop screen. The mirrored screen was also used to demonstrate the use of the feature viewer tool before the experiment.

5.3.6 Creation of Experiment Worksheets

In order to create a paired-samples experiment and reduce the effect of other variables, such as experience or task selection, we decided to ask each participant to perform each task twice. Once the task would be performed using the feature test tool, and once using only text search. We wanted the two instances of each task to be as far apart as possible with the session. To do this, we created a method of preparing a randomized worksheet that was different for each participant. Before each experiment, we created a worksheet using the following method:

Create a random order for the 7 tasks.

Assign the first task to a group:

50% experimental / 50% control

For each subsequent task, assign it to a group:

90% opposite group to last task / 10% same as last task

This 90%/10% method prevented a strict A/B alternation, by allowing two successive tasks in the same group to occasionally occur.
After the first seven tasks were listed, the second half of the list was created by copying the first seven tasks and selecting the opposite group (experimental or control) for each one of them, but preserving the order of the tasks.

In this way, each participant session had a list containing 14 tasks, creating 7 experimental/control pairs. In the space between any two instances of the same task there would be six other tasks to complete, effectively erasing the short-term memory of the first task in the pair. (In fact, this worked well; in over 100 pairs we saw only once evidence of a participant remembering information from the first occurrence of a task.)

A sample session worksheet is shown in Appendix B.

5.4 Experimental Participants

Test subjects were drawn from adult volunteers who have at least two years of professional programming experience in an agile environment, and are familiar with the purpose of the software system being examined. The majority of the subjects were programmers at a large commercial IT organization, though two worked for software vendors and three were involved in scientific programming endeavors elsewhere. The time required for each subject to participate was approximately one hour for feature location tasks.

Participants were asked for their experience levels, in years of programming experience, during the experimental session. These levels were recorded along with the location and session data. Note that participant skill selection was not critical because we could normalize by participant if necessary, and because each participant contributed to matched experimental and control samples.
We engaged 8 participants for the pilot tests, 4 participants for the GUI focus sessions, and 16 participants for the final study.

5.5 Experimental Protocol

This phase consisted of actually collecting data with participants in data collection session. All experimental sessions followed the same procedure for initiation, data collection, and conclusion.

5.5.1 Session Initiation

Experimental sessions were conducted in small conference rooms, with two occupied chairs on opposite sides of a work table. An LCD monitor (approximately 21”) was placed facing one side of the table along with a keyboard and mouse. The monitor, keyboard, and mouse were attached to the laptop using a mirrored display mode.

The feature viewer software was started on the laptop and loaded with the source code and feature location data as described above. The participant was given some instruction on how the various capabilities of the software worked, and given time to become familiar with the software. As much time as needed was allowed. Participants were allowed to ask any questions they deemed appropriate.

5.5.2 Session Data Collection

The participant was then presented with each task by the investigator based on the randomized worksheet discussed in Section 5.3.6. For each of the fourteen tasks, the investigator read the task description, accompanied by instructions to either use or not use
the test-based feature location tool as appropriate. When the feature location tool was not used, the window for the tool was hidden before the task commenced.

For each of the fourteen tasks, the investigator recorded on the worksheet:

- Time to complete, to the nearest second.
- Participant confidence in the lines identified as the location, on a scale of 0 to 10.
- The lines identified by the participant for that feature, saved in a location data file coded to the participant and task ID.

5.5.3 Session Conclusion

At the end of each session, each participant was asked three questions:

- “How many years of professional development experience do you have?”
- “How realistic was this experience and these tasks, on a scale of 0-10?”
- “How useful was the test-based feature location tool, on a scale of 0-10?”

After answering all the questions and the observer put the data collection worksheet away, participants were allowed to ask questions about the research.

After each session, the session data were entered into a spreadsheet, and the location files for the session were saved in a data directory coded by session and participant ID.

The experimental session workflow, including post-session data processing, is shown in Figure 5.2.
Figure 5.2: Experimental session workflow.
5.6 **Data Processing**

In this section we describe preparing the data for statistical analysis.

5.6.1 **Data Assembly**

While data were initially recorded in Excel, a small utility in Python was written to read these tables into Python data files in an IPython data analysis session. Data sets were created for participants and for tasks. Location data files created during sessions were used to add participant-identified location data to each task record.

Sample participant and task records are shown in Table 5.2 and
Table 5.2. Example participant data records

<table>
<thead>
<tr>
<th>User</th>
<th>Experience</th>
<th>Realistic</th>
<th>Useful</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>wz</td>
<td>1</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>mt</td>
<td>30</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>su</td>
<td>25</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tl</td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>pretty realistic</td>
</tr>
</tbody>
</table>
Table 5.3. Example task data records, after combining with location data

<table>
<thead>
<tr>
<th>User</th>
<th>Feature</th>
<th>Group</th>
<th>Time</th>
<th>Confidence</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>9</td>
<td>1</td>
<td>135.0</td>
<td>8</td>
<td>[3312, 3313, 3314, 3315, 3316, 3317, 3318, 3319, 3320, 3321, 3322, 3323, 3324, 3325, 3326, 3327, 3328, 3329, 3330]</td>
</tr>
<tr>
<td>rt</td>
<td>2</td>
<td>2</td>
<td>29.0</td>
<td>9</td>
<td>[2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261]</td>
</tr>
<tr>
<td>be</td>
<td>B</td>
<td>2</td>
<td>11.0</td>
<td>10</td>
<td>[2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261]</td>
</tr>
<tr>
<td>be</td>
<td>C</td>
<td>1</td>
<td>172.0</td>
<td>6</td>
<td>[2324, 2325, 2326, 2327, 2328, 2329]</td>
</tr>
</tbody>
</table>

5.6.2 Computation of Normalized Times

For each participant, a mean-task-time-for-participant was computed averaging all time to complete all feature location tasks, in both experimental and control tasks.

For each feature, a mean-task-time-for-feature was computed averaging time to complete that feature location task for all participants, both experimental and control tasks.

For each task in each group, we calculated:

- A time-normalized-by-participant value, dividing time by mean-task-time-for-participant.
- A time-normalized-by-feature value, dividing time by mean-task-time-for-feature.
These normalized values could be organized into paired samples for analysis with the variable of user or feature normalized.

We found when doing data analysis that these data sets were not required and did not use them in final data analysis.

5.6.3 Location Quality Score

To compute the quality of the identified location for each task, a simple crowdsourcing algorithm was used.

For each feature, a line number histogram was created, showing all lines identified for that feature, by line. For instance, if a particular line was associated with feature A for every task, with 16 participants x 2 groups, it would have a line count of 32. If a line was never associated with a feature, a line count of zero. If a line was identified half the time, it would have a line count of 16.

An example of this process is shown in Figure 5.3, showing some lines of code surrounding a specific feature. For feature C (“prevent sending an invalid file”), the letters on the left indicate each time a participant selected that line as implementing feature C. Upper case ‘C’ indicates a selection in a task in the experimental group, and lower case “c” indicates a selection in a control group task.
Figure 5.3: Line selection occurrences for feature C.

For each feature, we treated any line that was selected in more than ½ of the feature location tasks for a feature as an accepted line for that feature. These lines are indicated visually in Figure 5.3 by a vertical marker indicating lines selected more than 16 (of 32 possible) times. There are six such lines.

The accepted line sets for all features were computed and kept in a table. An example of this data for feature C (“prevent sending an invalid file”) is shown in Figure 5.4.
Figure 5.4: Accepted location line set for feature C.

The computed accepted reference locations for each feature were reviewed by expert Python programmers and deemed to be reasonable locations for those features. All accepted feature location sets are included in Appendix C.

In a second pass, for each feature location task, the identified lines in that task were compared with the accepted reference lines for that feature. The initial location quality score given to the task was the percentage of accepted lines included in the identified location set.

The location quality score was then reduced by 2 points for every included line that was not in the accepted set. Our reasoning was that inclusion of extraneous lines wasn’t ideal, so it should be penalized. However, the primary mission was to find the location of the feature, so inclusion of the accepted reference line was much more important and more heavily weighted, and extraneous lines only carried a small penalty.

The Python code that computed this value is listed in Figure 5.5.

```python
2324   C  if not filename.startswith(root):
2325       C  return HTTPError(403, "Access denied.")
2326   C  if not os.path.exists(filename) or not os.path.isfile(filename):
2327       C  return HTTPError(404, "File does not exist.")
2328   C  if not os.access(filename, os.R_OK):
2329       C  return HTTPError(403, "You do not have permission to access this file.")

[2324, 2325, 2326, 2327, 2328, 2329]
```
def location_quality_score(location, accepted):
    # find percentage of lines in accepted
    # that are in the identified location
    k = 0.0
    for item in location:
        if item in accepted:
            k = k + 1.0
    coverage_of_accepted = k / len(accepted) * 100.0
    # subtract two points for extra lines
    extra_lines_deduction = (len(location) - len(accepted)) * 2.0
    score = coverage_of_accepted - extra_lines_deduction
    # can't get less than zero
    return max(score, 0)

Figure 5.5: Computing the location quality score.

This method yielded a location quality score for each task between 0-100, where 100 was an ideal result. It is understood that this is not an ideal measure for quality, but the goal was to generate a metric that could compare quality of two location sets, and this is sufficient for that task.

The data analysis and evaluation workflow is shown in Figure 5.6.
Figure 5.6: Data analysis and evaluation workflow.
5.7 Evaluation of Results

Null hypothesis testing was done by comparing experimental and control groups for time, normalized times, confidence, and location quality score. Evaluation was performed with a paired t-test from the Python stats library.

In addition, histograms and 2-d scatter charts were made to help in visualizing patterns in the data, and are presented in the result section.

5.8 Addressing Threats to Validity

A number of steps were taken to address potential concerns about the study.
### Table 5.4. Validity threats / risks and mitigation steps taken.

<table>
<thead>
<tr>
<th>Threat / Risk</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software evaluated is not realistic</td>
<td>Use software in commercial production use, developed under test-driven-development practices.</td>
</tr>
<tr>
<td>Experimental software is not realistic</td>
<td>Design software after real-world interfaces, use common GUI affordances; verify realism with participants.</td>
</tr>
<tr>
<td>Feature location method is not useful in real word</td>
<td>Simulate a real world experience; evaluate usefulness with controlled experiment.</td>
</tr>
<tr>
<td>Participant pool is not realistic for real world</td>
<td>Draw participants solely from professional software developers, currently being paid to write software.</td>
</tr>
<tr>
<td>Skill factors – experience, training, familiarity with subject</td>
<td>Collect experimental and control sample tasks in pairs from every participant; normalize by participant if required.</td>
</tr>
<tr>
<td>Features differ in difficulty of location</td>
<td>Collect experimental and control sample tasks in pairs from every feature; normalize by feature if required.</td>
</tr>
<tr>
<td>Time of day, participant fatigue, etc.</td>
<td>Collect experimental and control sample tasks in pairs at nearly the same time in the same session.</td>
</tr>
<tr>
<td>Participant learns in first task, affects second task in pair</td>
<td>Separate tasks in pairs by several (six) intervening unrelated tasks. Monitor for evidence of reuse of recalled information.</td>
</tr>
</tbody>
</table>
CHAPTER 6

Results

6.1 Feature Extraction

Feature location extraction from bottle.py was performed using the methods of Section 5.3.

• Source code for bottle.py was retrieved by repository cloning from https://github.com/defnull/bottle.

• Unit tests were run using the unittest.py framework, with coverage collected for each test. 304 tests were run in 38 seconds. 294 created code coverage events. 280 of these tests created covered lines in bottle.py. (The non-coverage-generating tests were skipped due to unavailable plug-in options.)

• The commit history was extracted using git log --format=fuller --reverse. At this point, 1328 commits had been performed.

• Commits were divided into iterations using historical data. Each multifile commit activity was treated as an iteration.

• Feature location extraction using unit tests was done using the method of Section 5.3.3.

Though most of the tests produced coverage traces in the bottle.py source, when those coverage traces were masked by the feature extraction method, many tests were
entirely masked. The majority of the tests, however, retained some coverage, and the area covered was much smaller. Two of the tests were discarded after masking due to coverage errors, and tests with zero coverage were discarded. Some descriptive statistics about the test coverage can be found in Table 6.1.

**Table 6.1. Descriptive statistics for test coverage set sizes.**

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Masked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of tests</td>
<td>294</td>
<td>294</td>
</tr>
<tr>
<td>Tests with coverage</td>
<td>280</td>
<td>159</td>
</tr>
<tr>
<td>Average of coverage set size</td>
<td>141</td>
<td>10</td>
</tr>
<tr>
<td>Std deviation of coverage set size</td>
<td>135</td>
<td>12</td>
</tr>
</tbody>
</table>

Clearly the masking process produced smaller coverage set sizes, at an average size that is easier to comprehend as a feature location suggestion. While not all tests yield usable coverage, enough tests do that there are tests covering a large part of the program. We were also interested in what proportion of the entire program was covered by the resulting coverage sets. Results are in
Table 6.2.
Table 6.2. Lines of code, non-comment code, and covered code.

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Masked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>3562</td>
<td>3562</td>
</tr>
<tr>
<td>Lines of non-comment code</td>
<td>3049</td>
<td>3049</td>
</tr>
<tr>
<td>Lines of covered code</td>
<td>1333</td>
<td>1333</td>
</tr>
<tr>
<td>Total size of all coverage sets</td>
<td>39360</td>
<td>2108</td>
</tr>
</tbody>
</table>

Two things are interesting here. First, the coverage seems insufficient, but when examining the code, it is found that many of the uncovered lines are multi-line quotes treated as document comments, metadata declarations, decorators, and other non-active code. When reading the listing of non-covered lines, very little active code is encountered except for optional plug-in code that was not tested (since we aren’t using those plug-ins.)

Second, the coverage number is the same for the initial unittest coverage and for the masked coverage, even though the total size of the masked coverage sets is much smaller. While this isn’t intuitive, it is clearly expected, because the only reason a line would be removed from a coverage set is that it was covered earlier by a previous test. So the masked coverage is as broad, but much thinner. The depth, the count of how many tests covered any given line, was clearly different before and after masking.

To investigate coverage depth, we looked at the two coverage sets, and computed for each coverage set the average and maximum depth of coverage. Results are in Table 6.3.
### Table 6.3. Depth of coverage – count of tests invoking covered lines.

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Masked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of tests</td>
<td>294</td>
<td>294</td>
</tr>
<tr>
<td>Tests returning coverage sets</td>
<td>280</td>
<td>159</td>
</tr>
<tr>
<td>Average number of tests invoking a</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>covered line</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of tests invoking a</td>
<td>142</td>
<td>6</td>
</tr>
<tr>
<td>covered line</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This means that before masking, the average line that was covered was invoked by an average of 29 tests. Assuming that the tests did not all test for the same feature, this means that directly corresponding unmasked unit tests by coverage to source code was a very ambiguous proposition. In fact, before masking, there was at least one covered line invoked by 50% of all coverage-generating unit tests. On the other hand, after masking, line correspondence to specific tests was very high.

### 6.2 Experimental Evaluation of Location Usability

Experimental data for the location usability study was collected as described in Section 7.5 for 16 participants over a course of approximately two weeks in February and March, 2014. Results are described in the following sections.
6.2.1 Participant Questions

After each session, participants were asked for their experience as a professional software developer, and for their subjective 0-10 rating of the realism of the experiment and the perceived usefulness of the feature search technique. Results are shown in Table 6.4.

Table 6.4. Participant question results

<table>
<thead>
<tr>
<th>Measure</th>
<th>N</th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experience (years)</td>
<td>16</td>
<td>1.0</td>
<td>35.0</td>
<td>18.1</td>
<td>10.2</td>
</tr>
<tr>
<td>Was it realistic? (0-10)</td>
<td>16</td>
<td>7.0</td>
<td>10.0</td>
<td>8.4</td>
<td>1.0</td>
</tr>
<tr>
<td>Was it useful? (0-10)</td>
<td>16</td>
<td>8.0</td>
<td>10.0</td>
<td>9.4</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Participants had a wide range of experience. Participants generally thought the test was helpful, and rated the feature search capability as very useful.

6.2.2 Statistical Task Results

The experimental variables, as described in Chapter 5, were:

- Time to complete, in seconds.
- Confidence in location, ranging from 0-10.
- Location quality score, ranging from 0-100.

**Descriptive Statistics**

During 16 sessions, with 14 tasks per session, 224 feature location tasks were performed. These consisted of 112 pairs of matching tasks, one each from the experimental and control group.
Initial descriptive statistics were computed for the experimental and control groups. These results are shown in Table 6.5 and Table 6.6.

**Table 6.5. Control (text search only) location task results**

<table>
<thead>
<tr>
<th>Measure</th>
<th>N</th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to complete (seconds)</td>
<td>112</td>
<td>18.0</td>
<td>390.0</td>
<td>88.6</td>
<td>64.0</td>
</tr>
<tr>
<td>Confidence in location? (0-10)</td>
<td>112</td>
<td>1.0</td>
<td>10.0</td>
<td>7.4</td>
<td>2.4</td>
</tr>
<tr>
<td>Location quality score? (0-100)</td>
<td>112</td>
<td>0.0</td>
<td>100.0</td>
<td>76.4</td>
<td>39.2</td>
</tr>
</tbody>
</table>

**Table 6.6. Experimental (with feature tool) location task results**

<table>
<thead>
<tr>
<th>Measure</th>
<th>N</th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to complete (seconds)</td>
<td>112</td>
<td>8.0</td>
<td>118.0</td>
<td>28.4</td>
<td>19.8</td>
</tr>
<tr>
<td>Confidence in location? (0-10)</td>
<td>112</td>
<td>3.0</td>
<td>10.0</td>
<td>8.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Location quality score? (0-100)</td>
<td>112</td>
<td>0.0</td>
<td>100.0</td>
<td>88.4</td>
<td>24.6</td>
</tr>
</tbody>
</table>

Results Graphs and Charts

To help visualize this, we created histograms for the experimental variables, shown in Figure 6.1, Figure 6.2, and Figure 6.3. In all figures, blue represents the control group, and green represents the experimental group. In Figure 6.1, the number of feature location tasks completing in less than 50 seconds was markedly higher for the experimental group.
In Figure 6.2, the confidence in location can be seen to be higher in the experimental group, with more experimental tasks completed with scores of 8, 9 or 10.
In Figure 6.3, higher location quality scores can be seen for the experimental group, with the control group having completed a higher number of low-scoring (i.e. less acceptable) feature location tasks.
To better visualize the overall performance benefit from the feature location tool, as realized by the experimental group, we created a scatter diagram showing, for control and experimental groups, a combination of the time and confidence scores. Overall, a better time and better confidence indicated a better result, and would be indicated by a concentration at the upper left (higher confidence, lower time) corner of the picture.

Figure 6.4 shows the scatter diagrams for the experimental and control groups. The experimental group shows a greater concentration of tasks in the faster time, more confidence section of the chart, indicating better overall performance.

**Figure 6.3: Location quality score result histogram.**
To generate a visual indication of the differences in the resulting identified locations, we generated a chart showing the line locations identified for each individual participant. In this chart, in Figure 6.5, we can see that while most of the locations identified are roughly equivalent, the control group has more errors and missed identifications than the experimental group.
Finally, we wanted to represent the relationship between the actual locations chosen and participant confidence in that selection. We observed that the locations were frequently very similar, which may be expected since human inspection was the last step in both experimental and control cases. However, the confidence in the location differed frequently even for identical locations. Figure 6.6 shows this.
Figure 6.6: Quality score vs. confidence for control and experimental groups.

Note that in Figure 6.6 even the highly scored location tasks (i.e. ones generally accepted as accurate) had lower confidence scores for the control group.

6.2.3 Hypothesis Testing

An independent-samples t-test was conducted to compare time-to-locate and confidence-in-location in text-search (control) and feature-search (experimental) conditions. In each case, the null hypothesis was that there was no significant difference in the control and experimental groups.

As discussed in Chapter 8, the results for the groups were first compared using an independent t-test. Results of this test are shown in
Table 6.7.
Table 6.7. Experimental hypothesis testing with independent t-test

<table>
<thead>
<tr>
<th>Measure</th>
<th>T-statistic (independent)</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to complete</td>
<td>9.51</td>
<td>&lt;0.00001</td>
</tr>
<tr>
<td>Confidence in location</td>
<td>-4.03</td>
<td>&lt;0.00001</td>
</tr>
<tr>
<td>Location quality score</td>
<td>-2.73</td>
<td>0.0068</td>
</tr>
</tbody>
</table>

These results allow us to claim that significant support was found for all three experimental hypotheses. In particular:

- There was a significant difference found in the time-to-locate scores for text-search (M = 89, SD = 64) and feature-search (M = 28, SD = 20) conditions; t(224) = 9.51, p < 0.0001. These results strongly reject the null hypothesis and confirm that feature-search lowers time-to-locate.

- There was a significant difference found in the confidence-in-location scores for text-search (M = 7.4, SD = 2.4) and feature-search (M = 8.5, SD = 1.5) conditions; t(224) = 4.03, p < 0.0001. These results strongly reject the null hypothesis and confirm that feature-search raises confidence-in-location.

- There was a significant difference found in the location quality score scores for text-search (M = 76.4, SD = 39.2) and feature-search (M = 88.4, SD = 24.6) conditions; t(224) = -2.73, p = 0.0068. These results strongly reject the null hypothesis and confirm at p=0.0068 that feature-search raises location quality scores.
Since the sample values are paired, we also computed a paired t-test, the results of which are very similar. Results of this test are shown in Table 6.8.

**Table 6.8. Experimental hypothesis testing with paired t-test**

<table>
<thead>
<tr>
<th>Measure</th>
<th>T-statistic (paired)</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to complete</td>
<td>10.1</td>
<td>&lt; 0.00001</td>
</tr>
<tr>
<td>Confidence in location</td>
<td>-5.5</td>
<td>&lt; 0.00001</td>
</tr>
<tr>
<td>Location quality score</td>
<td>-3.6</td>
<td>0.0005</td>
</tr>
</tbody>
</table>

Again, these results allow us to claim that significant support was found for all three experimental hypotheses. We did not compute t-tests for the normalized data as described in Section 5.6.2 as it was clearly not needed to establish significance.

Details of this analysis are found in an IPython Notebook session file, available at the repository discussed in section A.3 in the appendices.

Results are discussed in Chapter 7.
CHAPTER 7

Discussion and Conclusion

7.1 Review of the Research Problem

This research is focused on finding features in software based on code coverage methods using the method of *software reconnaissance*. Since this method requires masking tests to be successful, its use in modern software development has been limited, even though modern software development methods involve writing large numbers of feature-specific tests. In an environment of hundreds or thousands of invoking tests, there are no masking tests. The obvious conclusion is that masking tests are just not written as part of the process, thus rendering *software reconnaissance* useless in these circumstances. Recall Eisenberg’s comment:

> Test suites developed under TDD are a good source of exhibiting tests. However, they are *not* a good source of correlated non-exhibiting tests—indeed the TDD methodology does not call for the explicit construction of “tests that do not test feature X”. Because of this, we believe that techniques like Software Reconnaissance and Execution Slices cannot be used effectively with a typical test-suite developed under TDD, unless developers are willing and knowledgeable enough to develop correlated non-exhibiting tests themselves. (Eisenberg, 2005)

Our observation was that to achieve the benefits of the method, masking tests are actually not necessary. It would be sufficient to find *any* coverage that could serve to mask out the non-feature code in a feature test. An examination of the agile method showed model-based evidence that the *accumulation of coverage from features implemented in previous agile iterations* would, in theory, provide useful masking
coverage, enabling the use of software reconnaissance-like feature location methods for using agile feature tests to locate features.

The research problem, specifically, is to see if the method works, and if so, if it works well enough to enable better feature location performance in professional software developers locating features in unfamiliar code. To do this we first extracted features from a large program, and then created an experimental environment to test developer performance with and without a feature location tool based on those extracted feature locations.

7.2 Pilot Investigations

Initial pilot investigations were designed to verify a number of assumptions.

Creation of Coverage Capability

The first assumption was that it was possible to instrument modern application code in such a way that it would be possible to get a separate coverage trace for each unit test. If this could not be done, it would not be possible to treat unit tests as individual sources of coverage information. We initially planned to look at an ANSI C application, and developed an instrumentation package based on srcML and XML transformations, which worked well. In fact, we found srcML to be very capable in this regard.

However, when we found a suitable application for the experimental study written in Python, we found that modifying the existing unittest.py unit testing framework to use a coverage module was a straightforward exercise, basing our coverage module on intercepting Python’s trace capabilities. It was easy to run application unit tests and save
the location trace of each test into a data record, and then store all the locations to a file. These files could be easily opened by Python utility programs for later processing.

**Verification of Test Masking**

With coverage capabilities in place, our second effort attempted to verify the basic model of agile test masking by simulating a small iteration-based application development effort. We found that the basic idea of using a partial ordering of prior tests as masks worked well, but had some shortcomings. Specifically, we found that:

- Feature refactoring sometimes hid results if the feature was reimplemented as a part of a later, more advanced feature.

- Boundary conditions and guard tests sometimes cause inadvertent masking, since the boundary test got invoked negatively for earlier features.

- Incomplete testing may result in incomplete masks, which may allow non-feature code to appear in feature locations.

**Initial Conclusions**

In general, we felt after initial investigations that the method worked well enough to be generally useful, but not perfectly. This led to a research experimental hypothesis that the use of the feature was sufficiently beneficial that it would improve the performance of software developers engaged in feature location tasks in unfamiliar code.
7.3 Research Experiment Design

The basic research assertion was that the feature location data was sufficient to improve feature location performance. To evaluate this, we ran a controlled experiment with software developers. We asked participants to find features in code, considering feature location tasks performed using a tool based on the test masking method as the experimental group, and conventional text search methods as the control.

Selection of Study Application

Rather than study an artificial application, we used features derived from a commonly available, commercially popular open source application. For this purpose, bottle.py worked very well. It was an application based on a common idea – a web server – so that participants would grasp features definitions quickly. It was also developed using test driven development, so that both the source code and the agile artifacts, including a complete commit log, were available. The unit test framework was compatible with our earlier instrumentation of unittest.py from earlier code coverage investigations.

Selection of Participants

For participants, we used professional software developers. When we tried sessions with interested people who were not developers, the confusion from having little experience with software development environments and tasks overshadowed any effect from a feature location method. We concluded that for an experiment involving human
performance, we need to verify in advance that the subjects could execute the control task reasonably well.

Since we discovered this concern during some pilot activities, we were able to make corrections before the experimental study, and our qualification became to only use participants who were actually employed in the development of software. No further issues arose in this regard.

We also emphasize the value of cooperation with commercial software development organizations in obtaining data from these participants. We obtained permission to recruit developers partially from IT management interested in the research problem and in how work in this area might help software maintenance efforts. Clearly having a benefit for collaboration was key in obtaining assistance with participants.

*Feature Extraction Design*

Our feature extraction method, based on *git* commands to acquire source and commit log data and on Python programs for processing, worked well. We were able to use the unit testing modifications from the initial investigations with little modification to gather coverage. In general, Python proved a very capable method of doing this. With appropriate generalization of the method, it could be executed in a completely automated fashion against any application base where the source control method, agile artifact store, and unit test framework were completely understood. At this time, the entire process of acquiring source and data and generating masked feature coverage takes only a few minutes, making it acceptable for an operational part of the software development process.
Feature Viewer Design

Our initial design of the feature viewer, written in Python and Ttk UI tools, was straightforward to write. Reading the location data and displaying it on top of viewable source code posed no problem.

However, it turned out that a lot of small user interface problems impeded effective use of the experimental tool. After several exploratory sessions, we decided to engage prospective users in focus group reviews of the user interface. This worked well. We were able to discover a number of UI issues that were not relevant to the experiment and distracted or confused users. Some of these, like the code navigation bar, were features that might have been useful with additional training, but weren’t relevant to the experimental results. Others, such as an initial lack of keyword search in the feature tests, inhibited the use of the experimental data.

We also noted a number of user behaviors that were not productive, and we were able to tune the user interface to help with these. The most notable was the observation that if we presented a large number of tests without clearly indicating which ones were relative, users tended to follow a test’s identified feature coverage into the source code, and simply continue the search there. Having moved their attention to the source code, they never returned to the test collection to search for other possibilities. To compensate for this, we did two things:

- We initially tried telling users to be sure not to forget the rest of the tests.

  This was somewhat effective, but not entirely so.
We made all of the relevant tests immediately visible in the test area, highlighted with red icons, and expanded any subtrees containing tests.

This second approach worked well. By making the rest of the applicable tests obvious, the test selection continued to compete for attention when searching the source code wasn’t immediately rewarding. When the most useful test was the second or third related one, this method worked much better.

One other issue had to do with text searching in source code. While developing keyword search for the test tree, we also added it to the search capabilities for source code, as a side effect of using the same search code in both cases. After some investigation, we discovered that none of the environments the participants actually used had advanced search capabilities, and if the control environment was to simulate normal text search, the control capabilities should reflect that. We remodeled the source code search to do generally what Visual Studio does, with the exception of regular expression search, which our user group indicated they rarely use in feature search tasks. (In fact, unless required to program a regular expression, many of them rarely use them at all.)

Eventually we had a feature viewer that presented control and experimental capabilities reflecting the research problems. We restrained from adding additional improvements. We did test the viewer for some mock sessions to make sure we did not have obvious defects in the viewer. No defects were found during the experimental sessions.
Data Analysis and Design

Using IPython Notebook\(^3\) for interactive data analysis was extremely convenient. All data were read into notebook sessions where Python functions performed data processing to create the overall data sets. These data sets were then processed with statistics modules from the SciPy stats library, and plotted immediately with matplotlib, which created a very short, transparent, and convenient tool chain.

Overall, having a project that uses a single language for the vast majority of its computation made practical matters very convenient. All tools – editors, interpreters, debuggers, and libraries – were able to be used consistently during all stages of the project, which saved a great deal of time and effort.

7.4 Feature Extraction Results

The Python unit testing coverage capabilities worked very well for the tested application bottle.py, collecting coverage data with a minimum of fuss. The coverage data associated with each test was saved, along with the identifier of the test and test suite.

Acquiring the necessary agile artifacts from the commit log also turned out to be very easy, since the log format of Git repositories is easy to parse and includes all committed lines. By parsing the committed lines, it was possible to determine which tests

\(^3\) IPython Notebook is an interactive data analysis tool based on Python. It can be found at http://ipython.org/notebook.html. Anaconda is a very easy-to-install free distribution available at http://continuum.io/downloads.
originated in which commit events, and by mapping commits to iterations, it was possible to assign tests to iterations.

Once this was done, it was possible to partially order the tests by iteration and compute masks by accumulating test traces for each iteration into the mask data for subsequent iterations. Using this mask we isolated feature lines for over half of the unit tests, which was sufficient to identify locations for many features. The “focus” provided by the masking was able to associate these tests with relatively few lines of code, where their initial location traces frequently involved hundreds of lines.

Manual examination of the resulting feature identification by experienced Python programmers seemed to indicate that the features pointed to reasonable places in the code. This was a debugging step, as the experimental protocol was intended to assess the suitability of the feature location capability, regardless of our initial impressions.

For initial investigations, we left the feature traces associated with the keywords to be found in the titles of the tests and the titles of the test suites. No attempt was made to evaluate comments or other documentation describing the purpose of the tests, nor to parse the content of the identified lines of application code. Had this been done, the keyword store for each test might have been richer, but the existing keywords (from titles) seemed sufficient.

7.5 Usability Testing Results

Clearly the results from this investigation showed significant benefits to using the proposed feature location method in several experimental variables. This will be discussed more completely in Section 7.6.
There are several other observed results that are worthy of note.

First, participants overwhelmingly rated the experience as realistically recreating the experience, and to some extent the frustration, of a real-world feature location task. We frequently thought the participants lost track of the fact that they were in an experiment as they concentrated completely on the location task. It is important that the experiment was believable for the results to be extrapolated to ideas about how useful the method might be in a workplace capability.

Second, participants overwhelmingly rated the tool as useful. Subjectively, they found using the feature location tool rewarding. We heard comments like

• “After the third try, I didn’t want to use text search any longer”

• “I learned to trust the feature location tool”

In general, it would not have been promising to develop a tool that improved task scores but was so unpleasant to use that human factors preventing its use. The usefulness rating indicates that the tool was seen as a reasonable strategy for feature search.

Part of this success was due to focus group efforts to determine how the tool was presented and used, in order to optimize the strategies that participants employed when using the tool. It was an object lesson in how making a good tool that is difficult to use results in an undesirable outcome. Good technology needs good presentation.

Finally, we did notice a number of other test strategies employed by experienced developers when required to use text search. For instance, a number of developers searched using a general term until they encountered a more specific term that looked promising. When that happened, they switched to the new term immediately. For
instance, a developer looking for cookie encoding may start out looking for “encode” and
at some point see a reference to “cookie_encode” – and then immediately switch to
searching for the latter term. We wondered if we could find some way to support this
strategy by offering to search for the more specific term in tests.

Overall, we believed the experiment served its purpose in verifying that the
method was effective. We believe that the experiment, along with the initial feature
extraction effort, answered the proposed research questions.

7.6 Overall Research Results

We had several initial research questions. At the conclusion of the research, we
revisited the questions.

1. *Where in the agile process artifacts can useful data be found that can help us relate
tests to features of interest, and how does the model help us relate those data to those
features?*

We found that useful data for ordering the tests could be found in the commit log
and the iteration log. Extracting the required data from commit logs was very
straightforward, if the commit log included the changes for each commit. *Git* provides
this. It is assumed that other repositories have similar capabilities, or that such
capabilities can be constructed from version comparisons.

2. *Can a selected set of partially ordered tests, shown by artifact evidence to not include
the feature of interest, serve as a sufficiently useful non-invoking test?*

We found that the partial ordering of the tests, combined with the agile practices
of making sure that all features work at the end of an iteration and complete test
coverage, meant that we could use the accumulation of preceding tests as masks for subsequent ones.

3. *How well does such a method perform in actually locating features in code without manual intervention or “tuning” of the test selection process?*

Creating synthetic test masks using the techniques described produced effective feature location indicators, averaging about a dozen lines, in about half of the unit tests in the application. Since most features are addressed more than once in a suite of tests, we found that any of several tests in a suite could be used to identify some aspect of a feature and thus finding a feature.

We identified a number of factors earlier in discussing possible factors that might interfere with this method. It turned out that quite a few tests in our study were masked so completely that no feature lines remained. We have not investigated the individual reasons for these – that is out of the scope of this investigation – but it certainly seems like some of those possible scenarios may have occurred. Certainly something happened to make the performance less than perfect.

Even so, the research question was not about trying to make this method perfect. The question was about finding out whether the method, implemented in a straightforward way, could produce results sufficient to be useful in developer feature location tasks. Having come to the conclusion that the results were credible, we moved to that question.

4. *Can developers using those tests be shown to perform better in feature identification tasks than developers using traditional manual feature location methods?*
We evaluated this using a controlled experiment made reasonable attempts to create a realistic situation, which was confirmed by participants, and to eliminate confounding variables like feature complexity and participant skill.

Under these conditions, and for the software under test, significant differences were found in the performance of software developers engaged in feature location tasks. Times to locate were lower, confidence values for location results were higher, and location quality scores were higher.

Even if the location quality score seems like a relatively arbitrary way to assign values to location quality, it is not clear that there is any weighting of lines that would have resulted in a loss of significant difference. There is clearly a visual difference in the characteristics of selected locations and the location quality score is just a shorthand to represent that difference.

Given these significant differences in all three indices of developer performance, it seems reasonable to claim that we can answer this question affirmatively: developers using these tests for feature location guidance can be shown to perform better in feature location tasks.

7.7 Limitations and Mitigations

There were a number of places where the study accepted limitations on what we could study or the generality of the results we were collecting. In Section 7.8 we discussed threats to validity and planned mitigations. Here we review some of those limitations and other experimental concerns, and their corresponding mitigations.
7.7.1 Limitations of the Model

We noted earlier that there are a number of scenarios where the proposed test masking model would not be expected to work. We did not propose to address this shortcoming, but to determine if the level of efficacy was sufficient for the task of feature location. On other words, this limitation, and the seriousness of it, was a principle research question.

We also note that there are many agile software development efforts where the agile principles are not followed completely. Where that is the case, one can expect the assertions based on properties of the agile model to not be completely applicable. Again, the purpose of real-world research is to evaluate the effect of all of these contributions to loss of effectiveness in the extraction methods.

7.7.2 Limitations of Feature Extraction

We performed feature extraction on a single-file application, but we believe this is without loss of generality; all locations were flagged with file locations and the instrument could open several files, but it wasn’t necessary to do so with this application. (In fact, we did get data about coverage inside the unit tests themselves, which were in another file.) The only multi-file applications we could find with equivalent test quality were commercial applications. We may reproduce this effort using those applications, but it would be much harder to openly share results. We chose the one-file application over multi-file applications that were for these reasons considered less suitable.

We also believe the use of a Python application presents no special implications about limitation. The same data: unit tests, line coverage, file name, commits, etc. should
be available for any modern application language stored in a modern repository. For instance, we create coverage instrumentation for ANSI C and C# as part of our initial investigations, and would have had no problem preparing data-equivalent input files. So the selection of Python as a subject language does not, we believe, effect general applicability of the results.

Finally, we believe the application was of sufficient quality and complexity to present feature location challenges similar to a real-world application, and that the test driven development process was suitable to allow us to evaluate the agile artifacts that were central to the method.

7.7.3 Limitations of the Usability Study

The usability study evaluated professional developers who were unfamiliar with the application being studied, and who were not using the Python language frequently. It is possible that searching for features in a system where they were familiar with its structure and organization might yield different results. In this study we were specifically interested in modeling an unfamiliar situation, but perhaps variations on this might be interesting.

The usability study also took place in a real-world environment, where the variables of skill level and variations in feature complexity may have played a part. By designing the experiment so as to distribute these effects evenly across both experimental and control groups we hoped to minimize this effect. The very similar results of independent and paired t-tests indicate that these confounding variables did not interfere with the evaluation of the experimental variables.
Because we selecting participants from professional software developers, we believe the results are representative of the professional software development community. Using well-qualified participants was helpful in increasing the general applicability of the study.

7.8 Recommendations for Future Research

There are a number of opportunities for improving components in the feature extraction and end user tool chains. Here we discuss some areas of potential research.

7.8.1 Feature Extraction using Agile Artifacts

Undoubtedly improvements can be made in the feature extraction process. These generally divide into improvements understanding the initial model, incorporating alternate models, and tolerating error.

Model Improvements

The agile model is a simple one and the resulting extraction model and approach leads to reasonable results in a short time. However, there are some known shortcomings.

First, the model effectively ignores boundary conditions and minimal-case conditions that cause guard code to get injected into early tests and masked from late tests. For example, consider this conditional statement:

\[
\text{if } (x < 0): \ldots
\]

This code will get executed whether \(x < 0\) or not, because it is a guard condition for \(x\) being negative. The condition will be added to an initial test as a guard, and because
it is then (inappropriately) part of the mask, the line will not be highlighted with the code for processing the condition it was intended to check for.

It may be that the model could be refined to recognize the boundary and guard conditions and place them with the guarded code rather than in an earlier test.

Second, the model blurs coverage masks when tests that would effectively mask a feature are included in the same iteration, and thus not part of the computed mask. Test ordering doesn’t work when tests are created simultaneously in the same iteration. More sophisticated sub-iteration mask computation could be tried, or other more time-sensitive refinements to the model may help.

Alternate Models

Classical agile development is not the only way contemporary software developers are organizing software tasks. Variations on the agile method are numerous.

As long as the basic model tenets of application completeness and test coverage are present, the approach detailed here should work reasonably well. However it might be found that the approach could be improved from additional information from other processes.

Recently, Kanban, estimation poker, Water-Scrum-Fall, DAD, and other new concepts and processes have been defined. It may be that artifacts created in these activities, when fully understood, may be helpful in identify masks, or in enabling wholly new ways to identify features.
Tolerance to Informal Practices

As with many human activities, agile is a practiced rarely practiced ideally. Given that agile is not applicable all the time, it should be possible to determine if the feature location approach is useful or not useful at certain levels of agile accuracy, and this method used to determine what level of compliance is required for the intended benefit.

For instance, if it is determined that 90% coverage provides the necessary benefits, then pushing to close the last 10% of code coverage is not necessary. On the other hand, under some circumstances, that level of coverage may be acceptable and the effort can be spent elsewhere.

7.8.2 Feature Location Usability

The controlled experiment portion involved using a human subject to search for features under control and experimental circumstances. There are a number of possible improvements for usability of the feature location tools that merit additional investigation.

- Can the keyword search for candidate tests be made richer by including documentary contents and comments in the test?
- Can the search be made richer by including content from the lines being executed in the test coverage?
- Is the way the user interacts with the test list and the display of the identified coverage set optimal, or are there other factors to consider?
In general, we saw that the presentation of the tool made a big difference in the utility of the feature search method. It’s doubtful that it is being presented here in a fully optimized manner.

Additionally, there are some possibilities for additional improvements to the feature location process overall. We have already mentioned the search re-focusing behavior of experienced programmers. Would it be possible to pop up a list of related terms during the initial input of the search string. Upon typing in “cookie” they could also choose from any identifier in the current source containing “cookie”, which would, for instance, make the suggestion for “encode_cookie” visible immediately.

Finally, there are possibilities for using feature maps in new and innovative ways. For instance, doing things in reverse, it would be interesting to ask which features depend on a particular line of code. Combined with dependency analysis in the source code, one might ask which features depend on this line of code, or other lines that invoke this line. This would help determine what is being put at risk when a line of code is being changed, and suggest possible focused testing activities to mitigate risk from the change. Tracing dependency lines in both directions, it would be possible to ask what features might be put at risk that interact with a given feature being modified, and again plan more effective risk mitigation activities.

7.8.3 Hybrid Feature Location Methodologies

We believe that this feature location method shows great promise, but there are other feature location methods (see Chapter 2 for many of these), and there is a good
possibility that simultaneous application of several methods may have promise. Several basic approaches could be suggested:

- Can other feature location methods be used to specify general areas that can inform the masking process to get better results from test-based location extraction?
- Can test-based location methods provide good starting locations to methods that start with specific locations and work outward to the boundaries of the relevant sections? Could alternative methods start with locations identified with test-based methods and expand cleanly to the edges of functions and classes?
- Can test-based location provide a good start for dependency analysis of feature implementation? Starting with the identified lines, identifying dependencies called, and other placed referring the identified locations, one could construct a map of feature influence that traversed part of the application dependency map. Particularly as part of feature interaction analysis, this holds promise.

Finally, there are methods that have been used to visually present graphical representations of feature locations and dependencies. Using those methods in conjunction with hybrid location results may prove promising, especially in code comprehension tasks.
7.9 Conclusion

At the beginning of this research, we were motivated by the desire to find a more useful method to locate software features in a modern agile development process. We explored the process of software reconnaissance, a method for locating features by comparing code coverage from invoking and masking tests.

Noting that agile processes have many invoking tests and no intentional masking tests, we proposed that some other combination of coverage data might be useful as masks to create a feature location ability. To focus this idea, we asked some specific research questions.

• *Where in the agile process artifacts can useful data be found that can help us relate tests to features of interest, and how does the model help us relate those data to those features?*

• *Can a selected set of partially ordered tests, shown by artifact evidence to not include the feature of interest, serve as a sufficiently useful non-invoking masking test?*

• *How well does such a method perform in actually locating features in code without manual intervention or “tuning” of the test selection process?*

• *Can developers using those tests be shown to perform better in feature identification tasks than developers using traditional manual feature location methods?*
To answer these questions, an experimental research investigation was designed and successfully executed. We had several specific objectives.

- **To model agile methodology artifacts to create a method for selecting necessary tests for feature location via software reconnaissance.**
- **To verify the correctness of the model and method for extracting feature locations from source code and tests in agile projects.**
- **To verify that this method is sufficiently robust to be effective in enabling the task of locating features in real-world development efforts.**

This research showed that feature locations could be extracted from software using agile method artifacts and modifications to the software reconnaissance method. In particular, the investigation led to some specific conclusions.

- **It is possible to use the software reconnaissance method with a mask coverage created from something other than a deliberate masking test.**
- **One can create a reasonable mask by combining coverage from other tests in an agile unit test collection.**
- **Agile artifacts can be mined, using the agile model, to find an ordering that allows unit tests to be selected that create an effective mask.**
- **The resulting feature locations extracted from those coverage and mask artifacts create reasonable coverage results from a large number of tests.**
- **The resulting feature location information is sufficient to make a significant improvement in the performance of software developers engaged in feature location tasks.**
With this investigation, we have met our research objectives, and have addressed the research questions originally posed for this study. We found that features can be extracted from the combination of comprehensive unit tests and agile artifacts, and that even if this method has limitations, it can be done well enough to be useful to professional software developers in feature location tasks.

In practical terms, this opens up the methods of software reconnaissance to agile, test-driven agile projects, providing those projects with a practical method of feature location for use in application development, maintenance, feature risk analysis, product planning, and so forth.

This work also has promise for new research. When combined with other feature location methods, the accuracy of a hybrid method may be sufficient to use feature-based software modeling in reverse engineering and program comprehension. Feature location on a large scale may allow new methods of risk analysis, change estimation, and problem management. More accurate mapping of features to source code has the capacity to touch many areas of software engineering.

The research also contributes to a methodology for empirical analysis of the benefits of experimental software engineering performance tools, by demonstrating a method of creating a controlled evaluation of a tool to generate a statistical analysis of significant benefit.

Finally, we believe that the approach we have used – to experiment with a theoretical idea in a human context to see if it has practical application – has a great deal of promise as a research tool. There are many ideas that work less than mathematically
perfectly, and many good heuristics are left behind because of the existence of a few disproving use cases. The methodology of a controlled software engineering experiment may be used to demonstrate that, in the practice of professional software engineering, a reasonably good idea executed very well can be extremely effective indeed.
APPENDIX A

Program Listings

Listings of interest are included in this section. All code is available at a source repository described in Section A.3.

A.1 ANSI C Coverage

These files are used for instrumenting ANSI C code as discussed in Section 4.1.1.

coverage.awk

This program adds line and filename attributes to srcML tags for code coverage processing.

```awk
# This program adds line number attributes to sourceML xml files resulting from parsing
# ANSI C programs with the Source2ML utility.
#
function add_line_attribute(line,tag,n)
{
    old_tag = sprintf("<%s>",tag)
    new_tag = sprintf("<%s line="%d">",tag,n)
    gsub(old_tag,new_tag,$0)
}

#initialize line count
BEGIN {
    linecount = 0
}

#change the xmlns attribute to make xslt matching easier
/xmlns = /
    gsub("xmlns=",_xmlns_="",$0)
#
#process <function> tags
/<function>/ {
    add_line_attribute($0,"function",linecount);
}

#process <decl_stmt> tags
/<decl_stmt>/ {
    add_line_attribute($0,"decl_stmt",linecount);
}
```
#process <expr_stmt> tags
/<expr_stmt>/ {  
    add_line_attribute($0, "expr_stmt", linecount);
}

#process <return> tags
/<return>/ {  
    add_line_attribute($0, "return", linecount);
}

#process <if> tags
/<if>/ {  
    add_line_attribute($0, "if", linecount);
}

#process <while> tags
/<while>/ {  
    add_line_attribute($0, "while", linecount);
}

#process <do> tags
/<do>/ {  
    add_line_attribute($0, "do", linecount);
}

#process <condition> tags
/<condition>/ {  
    add_line_attribute($0, "condition", linecount);
}

#process <switch> tags
/<switch>/ {  
    add_line_attribute($0, "switch", linecount);
}

#increment line count, output result
/^/ {  
    linecount++  
    print $0
}
coverage.xslt

This XSLT transform adds additional code into structural components of a C program expressed in srcML to create equivalent instrumented code.

```xml
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
    xmlns:cpp='http://www.sdml.info/srcML/cpp'>
    <xsl:output method="xml"/>

    <xsl:template match="unit"><xsl:copy>
        <xsl:apply-templates select="@*"/>
        #include "coverage.h"
        <xsl:apply-templates select="node()"/>
    </xsl:copy></xsl:template>

    <xsl:template match="function/block">
        <xsl:copy>
            <xsl:apply-templates select="@*"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="decl_stmt">
        <block>__LINE("<xsl:value-of select="/unit/@filename"/>",<xsl:value-of select="/@line"/>);</block>
        <xsl:copy>
            <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="expr_stmt|if|while|switch|return">
    </xsl:template>

    <xsl:template match="do/condition">
        __LINE("<xsl:value-of select="/unit/@filename"/>",<xsl:value-of select="/@line"/>);while
        <xsl:copy>
            <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="do">
        do{__LINE("<xsl:value-of select="/unit/@filename"/>",<xsl:value-of select="/@line"/>);}
        <xsl:copy>
            <xsl:apply-templates select="@*"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="@*|node()">
        <xsl:copy>
            <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```
coverage.h

This header file is included in instrumented code to execute the instrumentation function. In this case, it just prints the event – it could store it in a database or log file.

#define __FUNCTION(s,i) printf("%s:%d\n",s,i);
#define __LINE(s,i) printf("%s:%d\n",s,i);

hello.c

This is an example C file to be instrumented, including a number of various control structures.

#include <stdio.h>

int main(int argc, char** argv)
{
    int x = 4;
    int y = 3;
    y = x = 2;
    printf("Hello, world!\n");
    if(argc)
        printf("Hello, if!\n");
    if(!argc)
    {
        printf("Hello, if!\n");
    } else
    {
        printf("Hello, else!\n");
    }
    while(argc-- > 0)
    {
        printf("Hello, while!\n");
    }
    do{
        printf("Hello, do/while!\n");
    }while(argc-- > -2);
    switch(argc)
    {
    case 1:
        printf("Hello, switch/case!\n");
        break;
    default:
        printf("Hello, switch/default!\n");
        break;
    }
    return 0;
}
**hello.instrumented.c**

This is a C file that has been instrumented using the approach discussed in Section 4.1.1.

```c
#include "coverage.h"
#include <stdio.h>

int main(int argc, char** argv)
{
    __FUNCTION__("hello.c",3);
    __LINE__("hello.c",5);int x = 4;
    __LINE__("hello.c",6);int y = 3;
    __LINE__("hello.c",7);y = x = 2;
    __LINE__("hello.c",8);printf("Hello, world!\n");
    __LINE__("hello.c",9);if(argc)
    {
        __LINE__("hello.c",10);printf("Hello, if!\n");
    }
    __LINE__("hello.c",11);if(!argc)
    {
        __LINE__("hello.c",13);printf("Hello, if!\n");
    }else
    {
        __LINE__("hello.c",17);printf("Hello, else!\n");
    }
    __LINE__("hello.c",19);while(argc-- > 0)
    {
        __LINE__("hello.c",21);printf("Hello, while!\n");
    }
    __LINE__("hello.c",23);do{
        __LINE__("hello.c",25);printf("Hello, do/while!\n");
    }while(argc-- > -2);
    __LINE__("hello.c",27);switch(argc)
    {
        case 1:
            __LINE__("hello.c",31);printf("Hello, switch/case!\n");
            break;
        default:
            __LINE__("hello.c",34);printf("Hello, switch/default!\n");
            break;
    }
    __LINE__("hello.c",37);return 0;
}
```
A.2 Python Coverage

These are files used for instrumenting Python code as discussed in Section 4.1.2.

coverage.py

This Python file implements the trace function that collects coverage data and tags it with test information. It also implements the base test type that invokes coverage collection.

```python
import sys
import unittest

_data = {}
_story = "undefined"
_number = 0

def _trace(frame,x,y):
    if (frame.f_code.co_filename[1] == 'U'):
        _data[(frame.f_code.co_filename, frame.f_lineno, _story, _number)] = 1
        # print frame.f_code.co_filename
        # print frame.f_lineno
        return _trace

def reset():
    _data = {}

def start():
    sys.settrace(_trace)

def stop():
    sys.settrace(None)

def raw():
    result = {}
    for (filename,line,story,number) in _data:
        if (filename.endswith("coverage.py")):
            pass
        else:
            result[(filename,line,story,number)] = 1
    return result

def _extend(line):
    return ((line[0:-1] + "%60s") % "")[0:59]

def _compare(x,y):
    return cmp(x[3],y[3])

def _likely(line):
    return -1

def list(filename,data):
    file = open(filename,'r')
    lines = file.readlines()
    file.close()
```
lines = map(_extend, lines)
likely = map(_likely, lines)
keys = data.keys()
keys.sort(_compare)
for (path, line, story, number) in keys:
    if (path.endswith(filename)):
        lines[line-1] = lines[line-1] + "# " + story + " "
    k = 1
for line in lines:
    print (" " + str(k)[-4:] + "", line
    k = k + 1

class TestCase(unittest.TestCase):
    story = "undefined"

    def setUp(self):
        global _story, _number
        _story = self.story
        _number = self.number
        start()
        unittest.TestCase.setUp(self)

    def tearDown(self):
        stop()
        global _story, _number
        _story = "undefined"
        _number = 0
        unittest.TestCase.tearDown(self)
testcalc.py

This file contains the feature defining tests for the simulated agile effort.

#testCalc.py
import unittest
import coverage
import calc

# Story 1 : Single Digits
# -- evaluate should raise ValueError exception for non-strings
# -- evaluateDigit should raise ValueError exception for strings where length != 1
# -- evaluateDigit should raise ValueError exception for non-numeric strings
# -- evaluateDigit should return accurate single digits
# -- evaluate should return accurate single digits
class SingleDigitTestCase(coverage.TestCase):
    story = "SD"
    number = 1
    iteration = 1

def testNonString RaisesException(self):
    self.assertRaises(ValueError,calc.evaluate,1)
    self.assertRaises(ValueError,calc.evaluate,{})
    self.assertRaises(ValueError,calc.evaluate,[])  
def testBadLength RaisesException(self):
    self.assertRaises(ValueError,calc.evaluateDigit,"")
    self.assertRaises(ValueError,calc.evaluateDigit,"00")
    self.assertRaises(ValueError,calc.evaluateDigit,"000")

def testNonDigit RaisesException(self):
    self.assertRaises(ValueError,calc.evaluateDigit," ")
    self.assertRaises(ValueError,calc.evaluateDigit," -")
    self.assertRaises(ValueError,calc.evaluateDigit,"x")

def testSingleDigit ReturnsCorrectValue(self):
    self.assertEqual(calc.evaluateDigit("0"),0)
    self.assertEqual(calc.evaluateDigit("9"),9)

def testEvaluate ReturnsCorrectValue(self):
    self.assertEqual(calc.evaluate("0"),0)
    self.assertEqual(calc.evaluate("9"),9)

# Story 2 : Positive Integers
# -- evaluate should return accurate values for positive integers from 1 to at least 7 digits

class PositiveIntegersTestCase(coverage.TestCase):
    story = "PI"
    number = 2
    iteration = 1

def testEvaluate ReturnsCorrectValue(self):
    self.assertEqual(calc.evaluate("0"),0)
    self.assertEqual(calc.evaluate("9"),9)
# Story 3 : Negative Integers
# -- evaluate should return accurate values for negative integers from 1 to at least 7 digits

class NegativeIntegersTestCase(coverage.TestCase):
    #story = "Negative Integers"
    story = "NI"
    number = 3
    iteration = 2

    def testEvaluateReturnsCorrectValue(self):
        self.assertEqual(calc.evaluate("-0"), -0)
        self.assertEqual(calc.evaluate("-1"), -1)
        self.assertEqual(calc.evaluate("-1234567"), -1234567)

    def testNoUnaryPlus(self):
        self.assertRaises(ValueError, calc.evaluate, "+4")
        self.assertRaises(ValueError, calc.evaluate, "+2")

# Story 4 : Positive Floats
# -- evaluate should return accurate values for positive integers from 1 to at least 7 digits
# -- evaluate should return accurate values for decimal points anywhere in the number
# -- evaluate should not allow more than one decimal points

class PositiveFloatsTestCase(coverage.TestCase):
    #story = "Positive Floats"
    story = "PF"
    number = 4
    iteration = 2

    def testEvaluateReturnsCorrectValue(self):
        self.assertAlmostEqual(calc.evaluate("0.0"), 0.0)
        self.assertAlmostEqual(calc.evaluate("1.0"), 1.0)
        self.assertAlmostEqual(calc.evaluate("3.14159"), 3.14159)

    def testDecimalPointsCanBeAnywhere(self):
        self.assertAlmostEqual(calc.evaluate(".314159"), 0.314159)
        self.assertAlmostEqual(calc.evaluate("314159."), 314159.0)

    def testOnlyOneDecimalPointAllowed(self):
        self.assertRaises(ValueError, calc.evaluate, "0.3.1459")
        self.assertRaises(ValueError, calc.evaluate, "0.31459.")
        self.assertRaises(ValueError, calc.evaluate, ".0.31459")

# Story 5 : Addition and Subtraction Expressions
# -- evaluate should return accurate values numbers added and subtracted
# -- evaluate should properly process left to right evaluation of multiple add/subtracts
# -- evaluate should allow unary negative integers but not allow additional operators

class AddSubExpressionTestCase(coverage.TestCase):
    #story = "Compound Add/Sub Expressions"
    story = "AS"
    number = 5
    iteration = 3

    def testEvaluateReturnsCorrectValue(self):
        self.assertAlmostEqual(calc.evaluate("1.0+2.0"), 3.0)
        self.assertAlmostEqual(calc.evaluate("1+2"), 3.0)
self.assertAlmostEqual(calc.evaluate("2.0-1.0"),1.0)
self.assertAlmostEqual(calc.evaluate("2-1"),1.0)

def testEvaluatesLeftToRight(self):
    self.assertAlmostEqual(calc.evaluate("3-2+1"),2.0)
    self.assertAlmostEqual(calc.evaluate("3-2-1"),0.0)
    self.assertAlmostEqual(calc.evaluate("3+2-1+3"),7.0)

def testAcceptNegativeNumbers(self):
    self.assertAlmostEqual(calc.evaluate("-3+1"),-4.0)

def testOnlyOnePlusMinus(self):
    self.assertAlmostEqual(calc.evaluate("-3+4"),1.0)
    self.assertAlmostEqual(calc.evaluate("3+4"),7.0)

# Story 6 : Multiplication and Division Expressions
#
# -- evaluate should return accurate values numbers multiplied and divided
# -- evaluate should properly process left to right evaluation of multiple operations
# -- evaluate should properly process precedence between mul/div and add/sub operations

class MulDivExpressionTestCase(coverage.TestCase):
    #story = "Compound Mul/Div Expressions"
    story = "MD"
    number = 6
    iteration = 3

def testEvaluateReturnsCorrectValue(self):
    self.assertAlmostEqual(calc.evaluate("1.0*2.0"),2.0)
    self.assertAlmostEqual(calc.evaluate("1*2"),2.0)
    self.assertAlmostEqual(calc.evaluate("1.0/2.0"),0.5)
    self.assertAlmostEqual(calc.evaluate("1/2"),0.5)

def testEvaluatesLeftToRight(self):
    self.assertAlmostEqual(calc.evaluate("3*2*2"),12.0)
    self.assertAlmostEqual(calc.evaluate("4/2/2"),1.0)

def testEvaluatesPriorities(self):
    self.assertAlmostEqual(calc.evaluate("3+2*2"),7.0)
    self.assertAlmostEqual(calc.evaluate("3-1/2"),2.5)

def testOnlyOneMulDiv(self):
    self.assertRaises(ValueError,calc.evaluate,"3**4")
    self.assertRaises(ValueError,calc.evaluate,"4/2")

try:
    unittest.main()
finally:
    coverage.list("calc.py",coverage.raw())
exit()
This is the final version of the library developed during the simulated agile effort.

```python
#calc.py
import string
cursor = 0
def evaluateDigit(c):
    if len(c) != 1:
        raise ValueError
    if not c.isdigit():
        raise ValueError
    return string.atoi(c)
def evaluateNumber(s):
    global cursor
    if not s[cursor] in string.ascii_letters + string.digits + string.punctuation + ' 

This is the final version of the library developed during the simulated agile effort.

```
op = s[cursor]
cursor = cursor + 1
if op == "+":
    value = value + evaluateMulDivExpression(s)
if op == "-":
    value = value - evaluateMulDivExpression(s)
return value

def evaluate(s):
    global cursor
    if (type(s) != type("")):
        raise ValueError
    if (len(s) < 1):
        raise ValueError
cursor = 0

return evaluateAddSubExpression(s)
A.3 Experimental Study Software

All software for the experimental study has been made available in a GitHub repository, along with sample data files and result files.

The available software includes the Python utilities used for feature extraction, the source for the feature viewer, and the IPython notebook files used for the various data analysis efforts.

Files may be found at: www.github.com/gregdelozier/dissertation

The author welcomes correspondence and inquiries at gregdelozier@gmail.com.

The software is licensed under the Gnu Public License, Version 2.0.
APPENDIX B

Experimental Instrument

Name: _________________________________________  Code: ____ Date: ________
B2: Using feature search, locate the code that saves an uploaded file
Time: ________  File: ____   Rating: ___________________________
F1: Using text search, locate the code that deletes a cookie
Time: ________  File: ____   Rating: ___________________________
G2: Using feature search, locate the code that handles header expiration
Time: ________  File: ____   Rating: ___________________________
E1: Using text search, locate the code that authenticates a request
Time: ________  File: ____   Rating: ___________________________
D2: Using feature search, locate the code that encodes a cookie
Time: ________  File: ____   Rating: ___________________________
C1: Using text search, locate the code that prevents sending an invalid file
Time: ________  File: ____   Rating: ___________________________
A2: Using feature search, locate the code that parses a date
Time: ________  File: ____   Rating: ___________________________
B1: Using text search, locate the code that saves an uploaded file
Time: ________  File: ____   Rating: ___________________________
F2: Using feature search, locate the code that deletes a cookie
Time: ________  File: ____   Rating: ___________________________
G1: Using text search, locate the code that handles header expiration
Time: ________  File: ____   Rating: ___________________________
E2: Using feature search, locate the code that authenticates a request
Time: ________  File: ____   Rating: ___________________________
D1: Using text search, locate the code that encodes a cookie
Time: ________  File: ____   Rating: ___________________________
C2: Using feature search, locate the code that prevents sending an invalid file
Time: ________  File: ____   Rating: ___________________________
A1: Using text search, locate the code that parses a date
Time: ________  File: ____   Rating: ___________________________
APPENDIX C

Accepted Feature Locations

These feature locations were extracted from the experimental workflow, as lines selected by more than half of the participants as implementing the feature.

A: Locate the code that parses a date

```python
2397  def parse_date(ims):
2398      """ Parse rfc1123, rfc850 and asctime timestamps and return UTC epoch. """
2399      try:
2400          ts = email.utils.parsedate_tz(ims)
2401          return time.mktime(ts[8]) + (ts[9] or 0) - time.timezone
2402      except (TypeError, ValueError, IndexError, OverflowError):
2403          return None
```

B: Locate the code that saves an uploaded file

```python
2244  def save(self, destination, overwrite=False, chunk_size=2**16):
2245      """ Save file to disk or copy its content to an open file(-like) object.
2246      If *destination* is a directory, :attr:`filename` is added to the
2247      path. Existing files are not overwritten by default (IOError).
2248      """
2249      if isinstance(destination, basestring): # Except file-likes here
2250          if os.path.isdir(destination):
2251              destination = os.path.join(destination, self.filename)
2252          if not overwrite and os.path.exists(destination):
2253              raise IOError('File exists.')
2254          with open(destination, 'wb') as fp:
2255              self._copy_file(fp, chunk_size)
2256      else:
2257          self._copy_file(destination, chunk_size)
```

C: Locate the code that prevents sending an invalid file

```python
2324  if not filename.startswith(root):
2325      return HTTPError(403, "Access denied.")
2326  if not os.path.exists(filename) or not os.path.isfile(filename):
2327      return HTTPError(404, "File does not exist.")
2328  if not os.access(filename, os.R_OK):
2329      return HTTPError(403, "You do not have permission to access this file.")
```

D: Locate the code that encodes a cookie

```python
2450  def cookie_encode(data, key):
2451      """ Encode and sign a pickle-able object. Return a (byte) string """
2452      msg = base64.b64encode(pickle.dumps(data, -1))
2453      sig = base64.b64encode(hmac.new(tob(key), msg).digest())
2454      return tob('!') + sig + tob('?') + msg
```
E: Locate the code that authenticates a request

```python
    def auth(self):
        """ HTTP authentication data as a (user, password) tuple. This
implementation currently supports basic (not digest) authentication
only. If the authentication happened at a higher level (e.g. in the
front web-server or a middleware), the password field is None, but
the user field is looked up from the `REMOTE_USER` environ
variable. On any errors, None is returned. """
        basic = parse_auth(self.environ.get('HTTP_AUTHORIZATION',''))
        if basic: return basic
        ruser = self.environ.get('REMOTE_USER')
        if ruser: return (ruser, None)
        return None
```

F: Locate the code that deletes a cookie

```python
    def delete_cookie(self, key, **kwargs):
        """ Delete a cookie. Be sure to use the same `domain` and `path`
settings as used to create the cookie. """
        kwargs['max_age'] = -1
        kwargs['expires'] = 0
        self.set_cookie(key, '', **kwargs)
```

G: Locate the code that handles header expiration

```python
    expires = HeaderProperty('Expires',
        reader=lambda x: datetime.utcfromtimestamp(parse_date(x)),
        writer=lambda x: http_date(x))
```
REFERENCES


