Leveraging the Agile Development Process for Selecting Invoking/Excluding Tests to Support Feature Location

Gregory S. DeLozier
Kent State University
Computer Science
Kent, OH USA
gdelozie@kent.edu

Michael J. Decker
Bowling Green State University
Computer Science
Bowling Green, OH, USA
mdecke@bgsu.edu

Christian D. Newman
Rochester Institute of Technology
Software Engineering
Rochester, NY, USA
cnewman@se.rit.edu

Jonathan I. Maletic
Kent State University
Computer Science
Kent, OH USA
jmaletic@kent.edu

ABSTRACT

A practical approach to feature location using agile unit tests is presented. The approach employs a modified software reconnaissance method for feature location, but in the context of an agile development methodology. Whereas a major drawback to software reconnaissance is the identification or development of invoking and excluding tests, the approach allows for the automatic identification of invoking and excluding tests by partially ordering existing agile unit tests via iteration information from the agile development process. The approach is validated in a comparison study with industry professionals, where the approach is shown to improve feature location speed, accuracy, and developer confidence over purely manual feature location.

CCS CONCEPTS

• Software and its engineering → Software creation and management

KEYWORDS

Software reconnaissance, invoking tests, excluding tests, feature location

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 [1]. Transfer of experienced developers and architects means retained program knowledge is lost [2]. Typical manual methods of feature location become unwieldy and impractical as program size grows [2].

Perfection in feature location is not always necessary. When developers attempt to locate a feature in code, they often search for an initial starting point where part of the feature is implemented, and then follow code from that point [3]. Further, developers tend not to use tools for this activity [4] Location of that initial point has been found to consume about half of the time of the feature location activity [3]. A method that could find this initial approximate location would be very helpful.

One of the first approaches proposed for feature location is called software reconnaissance [5]. In software reconnaissance a programmer provides feature-invoking tests that exercises the feature, and similar feature-excluding tests that avoids invoking the feature. Execution traces for these tests are collected. Code executed by the invoking tests but not by the excluding tests implements the feature. This provides a precise location, if tests are chosen carefully. However, the construction of invoking and excluding tests is non-trivial and the quality of the results are heavily dependent on proper test selection [6]. Several techniques exist to attempt to mitigate the test selection problem [6][7][8], however no automated method exists to select feature-excluding tests that are comprehensive enough to mask non-feature code and still not invoke the feature.

In current agile methods (i.e., test driven development (TDD) approaches) 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 [7, 8]. Efforts have been made to mine these agile artifacts for feature location. For instance, the textual labels of changesets have been found to correspond with features [9] and blending the labels of these artifacts with source code enhances feature location [10]. We can use the artifact labels similarly to location the times of feature implementations in the software history timeline, and to locate tests that correspond to specific feature implementations.

Once we have identified feature development time, we found that useful test selection methods can be constructed for agile software development projects by using these artifacts. By identifying a time prior to feature implementation, we can (in theory) 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 as test selection criteria, the method produces accurate results without requiring manual test selection.

Several different agile processes for software development are in widespread use; each emphasizes different aspects of agile methods. Here, we explicitly define a methodology to be agile if it follows three basic principles:

1) Development based on addition of small, well defined features with defined periods of time for implementing these features (i.e., iterations or sprints).

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00

2) Each feature is defined and verified by a collection of one or more 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. These tests are often written before writing the implementing code, but here it is only required that the tests exist and pass before the feature is considered complete, by completely exercising the feature.
3) Continuous integration of code is performed as it is checked into a repository. This includes performing all tests that have been defined to date, requiring all tests to pass.

This is a fairly well accepted definition within industry practice. One of the authors works at a large company where agile processes (as defined above) have been used for more than 10 years and applied to its entire code base (>100 MLOC). Since the basic agile methodology is a process executed under varying conditions, a feature location method requiring perfect execution of the process is of limited use. We evaluate the approach on a software system developed under agile conditions and find that, when compared to purely manual feature location, our method is sufficiently robust, and that it produces improvements in the quality and speed of feature location tasks even if the agile processes are followed incompletely or inaccurately to a degree.

The contributions of this paper are as follows: 1) Creation of a model of agile methodology and a method for selecting necessary tests for feature location via software reconnaissance. 2) Verification of the model, amethod for extracting feature locations from source code, and a method for extracting tests in agile projects.. 3) Verification that the method is sufficiently robust to be effective in enabling the task of locating features in real-world development efforts.

The rest of the paper is laid out as follows. Section 2 explains our model for extracting agile features. Section 3 presents the evaluation, and Section 4 Presents the results. Finally, threats to validity, related work, and conclusion/future work are in Section 5, 6, and 7 respectively.

2 EXTRACTING AGILE FEATURES

In this section, we discuss a model for extracting agile features. This model is then used to present the approach to feature location.

2.1 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_1…C_n where change C_i is the change that changed program P_{i-1} into program P_i. Let F represent the series of features or desired behaviors F_1…F_n, initially implemented by these changes, where F_i is the feature implemented by C_i. Let T represent the series of test suites T_1...T_n that test (i.e. verify the correctness and presence of) the features F_1...F_n. A change set S_i = [C_i, F_i, T_i] consists of the change C_i, the implemented feature F_i, and the verifying test suite T_i. Finally, let D represent the series of dates D_0...D_m, starting and ending the time periods (i.e. iterations or sprints) during which changes are made to add and test features in the program.

2.2 Modeling Agile Practices

As discussed in the introduction, 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.

Rule 1. Development is based on addition of small, well defined features with defined periods of time (iterations or sprints) for implementing these features. These periods of time may contain one or many commits to the code repository, but they must definitely end with a known commit event, and it must be possible to identify the set of commits performed during that period.

For some time period ending at D_k, let the changes, features, and change sets completed during this time be called S_p...S_r. For all q in p...r, the following holds: 1) On date D_{k-1}, no evidence of change set S_q is in the repository. 2) On date D_k, change set S_q is completed, tested and in the repository.

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

For some time period ending at D_k, let the changes, features, and change sets completed (and committed) during this time be called S_p...S_r. For all q in p...r, the following holds at time D_k: 1) for change set S_q = [C_q, T_q, F_q], 2) test T_q completely exercises feature F_q.

Note that 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.

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

When change C_p is introduced, resulting in P_p, then T_p exercises F_p. The continuous build rule means that the following holds at this time: For all q in 1…p-1, T_q exercises F_q for all P_x, x in (q..p). In essence, all previous tests continue to test the feature it was written to test, in all subsequent versions of the program.

Note also that a forward mapping must exist from T_q to F_q so that a test failure in continuous build can identify which feature is broken, and that we assume that a search of this forward mapping can create a mapping from F_q to T_q. This is discussed in more detail in Section 2.3.

2.3 Mapping Tests to Features

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. This set of properties can be used to map tests onto features.

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 [11]. 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.

It is also possible to map tests to features based on embedded documentation in the code (such as Python *docstrings*) or on repository commit messages, or based on being committed to the repository in the same commit action

as a documented feature. Any method of mapping is suitable as long as it is consistent and predictable. If such a mapping method does not exist, the tests are of limited value to developers since a test failure during integration must reveal what feature is failing. As long as that mapping is reversible (mapping from test to feature implies that one can map from feature to one or more tests) then the mapping can be used for the method we are developing.

2.4 Invoking Tests

Rule 1 states that at any D_k, 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.

By definition, an invoking test, T_q, is a test that completely exercises feature F_q and continues to completely exercise features F_q 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 in different locations), 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 is retired from the list of active tests, 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 to be delivered differently, then by ***Rule 2***, test T_q will be modified accordingly to create the current definition of 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 abstracts transactions so that the details of these lists and mappings for a particular software system are isolated from the logic of identifying the tests.

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.

After identifying the invoking test, the test is executed within the software system and the code coverage set collected. A software abstraction easily allows 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 are created on demand, but rather created during 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 (e.g., daily build).

2.5 Excluding Tests

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 do not need a single explicit test, but rather a method to produce the excluding coverage set.

Let V be a coverage set consisting of all the lines of code from P that are executed in a particular scenario or test. Specifically, V_q is the set of lines of code, the coverage set, generated from running test T_q exercising feature F_q. Note that coverage sets can be combined, where $V_{i...j}$ consists of the union of sets V_i...V_j. In other words, any line executed by any test T_i...T_j is included in $V_{i...j}$.

Recall, from ***Rule 1***, that at date D_{k-1}, change sets S_p...S_r do not yet exist. Let P_{p-1} represent the program state at date D_{k-1} and let S_q be a change set in S_p...S_r. Since, by ***Rule 1***, all features in P_{p-1} are complete, no test T_0...T_{p-1} can require or invoke any feature in F_q in S_q.

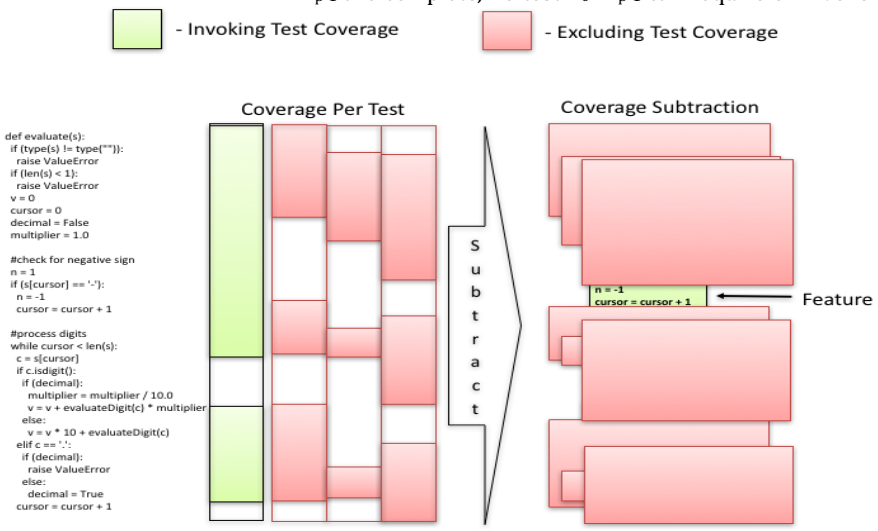

Figure 1: Illustration of the feature location process. Coverage information is gathered for the invoking test(s) and excluding tests (unit tests for previous iteration features). The union of the excluding tests coverages are subtracted from the invoking test, resulting in the lines of code associated with the feature.

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 and a test T, where by ***Rule 2***, each T completely tests F. The tests T_0...T_{p-1} completely test all features F_0...F_{p-1} in P_{p-1}. The coverage set $V_{0...p-1}$ (as the union of V_0 ...V_{p-1}) covers all features F_0...F_{p-1}.

Since the coverage set $V_{0...p-1}$ completely covers all features preceding S_q (and F_q), it serves as a reasonable choice for a masking set for excluding features in F_q.

Another way to look at this is to consider V_p as a test executing many features including the feature of interest. Ideally, the execution of any feature defined prior to the feature of interest (e.g., F_{p-1}) will 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 V_0 ...V_{p-1} do not serve to mask features created in the same iteration, i.e., it does not mask any F_q in F_p...F_r. Formally, if the feature F_q uses $F_{q'}$ (q' in p..r, q' ≠ q) then $V_{0...p-1}$ will not serve to exclude $F_{q'}$ from the test

for F_q. 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 possible that in a feature location task, narrowing to one or two features out of a large body of code is still helpful. We investigated this further in the evaluation.

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 is a subject of the evaluation; to determine if such coverage gaps are widespread and if they significantly detract from the usefulness of the approach. Figure 1 illustrates this process.

2.6 Creation of Excluding Coverage Sets

Since the running of tests is time consuming, it is 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 2.

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 EVALUATION

The goal of this evaluation is to assess the effectiveness of the proposed approach for developers working in real-world agile environments and projects. Formally, we seek to answer the following research question:

Does the proposed approach of using iteration history to form invoking and excluding tests be used effectively by developers for feature location tasks in real-world agile environments and projects?

To answer the research question, a controlled, within-participant experiment was designed to compare the effectiveness of developers in feature location tasks when using: 1) a feature viewer implementing the proposed method 2) manually with only text search. In the design of the controlled experiment and feature viewer, pilot and focus group studies were conducted to improve study design, remove deficiencies in the feature viewer, and remove other confounding variables. We break down the general research question into the the following research sub-questions answered by the controlled experiment:

- ***RQ1:*** *Do software developers using the tool find features faster than developers not using the tool?*
- ***RQ2:*** *Do software developers using the tool more accurately identify feature locations compared to programmers not using the tool?*
- ***RQ3:*** *Are software developers using the tool more confident that they answered correctly, as compared to developers not using the tool?*
- ***RQ4:*** *Do developers feel that the feature viewing tool is useful?*
- ***RQ5:*** *Would developers use the tool in real-world situations?*

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

Figure 2: Algorithm for producing code coverage for iterations.

3.1 Data Preparation

In order to have a realistic experiment and adequately answer the research questions, it was determined that open-source software exhibiting characteristics of a commercial application software be selected. In particular, the following criteria were used:

- Currently used 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 investigating several projects, the web server bottle.py was selected (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 (a manageable amount for a human experiment) presented in one large file for ease of deployment. The system also includes several hundred unit tests, which are suitable for the feature location method described in a later section.

After cloning the repository, the commit history (1328 revisions) was analyzed to assign each unit test to an iteration. Since *bottle* lacks formal iterations, each commit is treated as an agile iteration. Then, the unit tests were run using *unittest.py* (modified to collect coverage information), and the coverage information for each test was saved to a data file. Next, the coverage data and test iteration data were used to calculate the masked test coverage for each test according to the approach described previously. The masked coverage (i.e., the feature location for that test's feature) for each test is saved in a data file, indexed by the fully qualified test name.

In total there are 304 tests (ran in 38 seconds) and 294 created code coverage events. 280 of these tests created covered lines in *bottle.py.* (The non-coverage-generating tests are skipped due to unavailable plug-in options to *bottle*.) Table 1 shows the descriptive statistics about the test coverage.

Most tests produced coverage traces in the *bottle.py* source, when those coverage traces are masked by the feature extraction method, many tests are entirely masked. The majority of the tests, however, retained some coverage, and the area covered is much smaller. Two of the tests are discarded after masking due to coverage errors, and tests with zero coverage are discarded.

We are also interested in what proportion of the entire program was covered by the resulting coverage sets (see Table 2). Two things are of interest. 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 is 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 is not intuitive, it is clearly expected, because the only reason a line would be removed from a coverage set is that it is 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, is 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 (see Table 3).

This means that before masking, the average line that is covered is 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 is a very ambiguous proposition. In fact, before masking, there is 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 is very high.

3.2 Feature Identification Tasks

The feature location tasks for the experiment are selected based on the results of focus groups to identify the major features of web servers and then creating a question about each of seven identified major features. The feature location tasks are listed in Table 4.

3.3 Participants

Sixteen participants, from Progressive Corporation, were drawn from adult volunteers with at least two years of professional programming experience in an agile environment. Progressive uses an agile process enterprise wide. All participants are familiar with the purpose of *bottle*. The majority are programmers at a large commercial IT organization, two worked for software vendors and three were involved in scientific programming endeavors elsewhere.

3.4 Experimental Setup

To conduct the experiment, the feature viewing software is presented 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 are both loaded into the feature viewing 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.

Table 1: Descriptive statistics for test coverage set sizes.

	Initial	Masked
Total number of tests	294	294
Tests with coverage	280	159
Average of coverage set size	141	10
Std dev of coverage set size	135	12

Table 2: Lines of code, non-comment code, and covered code.

	Initial	Masked
Lines of code	3562	3562
Lines of non-comment code	3049	3049
Lines of covered code	1333	1333
Total size of all coverage sets	39360	2108

Table 3: Depth of coverage – count of tests invoking covered lines.

	Initial	Masked
Total number of tests	294	294
Tests returning coverage sets	280	159
Average number of tests invoking a covered line	29	1
Maximum number of tests invoking a covered line	142	6

Table 4: Tasks done by participants during the experiment

Code	Task
A	Locate the code that parses a date
B	Locate the code that saves an uploaded file
C	Locate the code that prevents sending an invalid file
D	Locate the code that encodes a cookie
E	Locate the code that authenticates a request
F	Locate the code that deletes a cookie
G	Locate the code that handles header expiration

In order to create a paired-samples experiment and reduce the effect of other variables, such as experience or task selection, we asked each participant to perform each task twice (once using the feature viewing tool and once manually using only text search). We wanted the two instances of each task to be as far apart as possible within the session. To do this, we created a method of preparing a randomized worksheet that is different for each participant. The worksheet is created using the following method:

- Create a random order for the 7 tasks
- Assign first task to group: 50% experiment/50% control
- Assign each remaining task to group based on last task: 90% opposite group / 10% same group

The 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 are listed, the second half of the list is 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 were 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 a single case of a participant remembering information from the first occurrence of a task.

For the experiment, we considered the control group to be the collection of answers collected without using the feature viewing tool, and the experimental group to be the collection of answers answered with the use of the tool.

3.5 Experimental Protocol

The following describes the procedure followed for each participant. First, the participant is instructed on how to use the feature viewing tool and given time to familiarize themselves. For feature location tasks, the participant is given as much time as needed (~1 hour on average). During the experiment, the participant is permitted to ask any questions they deemed appropriate.

The participant is then presented with each task by the investigator based on the randomized worksheet (subsection 0). For each of the fourteen tasks, the investigator reads 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 is not used, the window for the tool is hidden before the task commenced. For each of the fourteen tasks, the investigator records 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.

A post-experiment questionnaire was asked at the end of each session. The questionnaire consisted of the 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?"*

3.6 Measurements

A simple metric is used to assess the quality of the identified locations for each task. The metric, *Location Quality Score* (*LQS*), is based off the distribution of the results. For each feature, a line number histogram is created, showing all lines identified for that feature, by line. For instance, if a particular line is associated with feature A for every task, with 16 participants x 2 groups, it has a line count of 32. If a line is never associated with a feature, a line count of zero. If a line is identified half the time, it has a line count of 16.

An example of this process is shown in Figure 4 which illustrates the lines of code selected by participants for a specific feature. The example is for the feature "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.

```
CC          |               ||            original filename is used
CC          |               ||         :param charset: The charset t
CC          |               ||            mime-type. (default: UTF-
CC          |               ||     """
CCC         |               ||
CCCCCCCCC   |               ||     root = os.path.abspath(root) + os
CCCCCCCCC   |               ||     filename = os.path.abspath(os.pat
CCCCCCCCC   |               ||     headers = dict()
CCCCCCCCC   |               ||
cCCCcccCCcCcCC|CcCCcCC        ||     if not filename.startswith(root):
cCCCcccCCcCcCC|CcCCcCC        ||         return HTTPError(403, "Access
CcCCCcccCCcCcC|CCcCCccCCcC    ||     if not os.path.exists(filename) o
CcCCCcccCCcCcC|CCcCCccCCcC    ||         return HTTPError(404, "File d
cCCCcccCCcCcCC|CcCCcCC        ||     if not os.access(filename, os.R_O
cCCCcccCCcCcCC|CcCCcCC        ||         return HTTPError(403, "You do
cCCC        |               ||
cCC         |               ||     if mimetype == 'auto':
cC          |               ||         mimetype, encoding = mimetype
cC          |               ||         if encoding: headers['Content
cC          |               ||
cC          |               ||     if mimetype:
```

Figure 4: Line selection occurrences for feature "prevent sending an invalid file".

```
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
    #subtract two points for extra lines
    #not in the accepted location lines
    extra_line_deduct =
            (len(location) - len(accepted)) * 2
    score=coverage_of_accepted-extra_line_deduct
    #can not get less than zero
    return max(score,0)
```

Figure 5: Method for computing location quality score.

For each feature, we treat any line that is 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 4 by a vertical marker indicating lines selected more than 16 (of 32 possible) times. There are six such lines.

The computed accepted reference locations for each feature were reviewed by expert Python programmers and deemed to be reasonable locations for those features. In a second pass, for each feature location task, the identified lines in that task are compared with the accepted reference lines for that feature. The initial location quality score given to the task is the percentage of accepted lines included in the identified location set.

The location quality score is reduced by 2 points for every included line that is *not* in the accepted set. Our reasoning is that inclusion of extraneous lines is not ideal, so it should be penalized. However, the primary mission is to find the location of the feature, so inclusion of the accepted reference line is much more important and more heavily weighted, and extraneous lines only carried a small penalty. Figure 5 gives the method for calculating the location quality score. This method yielded a location quality score for each task between 0-100, where 100 is an ideal result. It is understood that this is not an ideal measure for quality, but the goal is to generate a metric that could compare the quality of two location sets, and this method was satisfactory for that task.

4 RESULTS

Null hypothesis testing is done by comparing experimental and control groups for time, confidence, and location quality

score. The Python *stats* library is used for analyzing the results.

4.1 Post-questionnaire Results

The descriptive statistics for the post-questionnaire are are shown in Table 5. Participants had a wide range of experience. Participants generally felt the test was realistic (positive support for ***RQ5***), and rated the feature search capability as very useful (positive support for ***RQ4***).

4.2 Experiment Results

Initial descriptive statistics are computed for the experimental and control groups. These results are shown in Table 6. In summary and on average, the experiment group completed feature locations tasks quicker, more confidently and more accurately than the control group. Furthermore, there is less variation between members of the experiment group than in the control group (as seen by the smaller standard deviation).

Histograms for the experimental variable are given in Figure 6 thru Figure 8. In all figures, blue represents the control group, and green represents the experimental group. In Figure 6 the mean time to completion is clearly lower for the experimental group. Participants were asked to rank their confidence in their identification of the location of the feature from 0-10. In Figure 7 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 8 higher location quality scores (as described in section 3.5) 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.

A paired-samples t-test is used to compare time-to-locate and confidence-in-location in text-search (control) and feature-search (experimental) conditions. The t-test is selected as the sample size is sufficiently large to apply the Central Limit Theorem. Additionally, the t-test is sufficiently robust even in cases of non-normal and/or ordinal data [12]. In addition, we also ran Wilcoxon signed-rank test. We also found statistical significance in all measures. In each case, the null hypothesis is that there is no significant difference in the control and experimental groups. Results of this test are shown in Table 7 and Table 8.

Table 5: Participant question results

Measure	Min	Max	Mean	SD
Experience (years)	1.0	35.0	18.1	10.2
Was it realistic? (0-10)	7.0	10.0	8.4	1.0
Was it useful? (0-10)	8.0	10.0	9.4	0.7

In all cases, we are able to reject the null-hypothesis in favor of the approach. Furthermore, the experimental conditions had a very large effect on time to complete, and medium effect on confidence in location, and a small effect on location quality score. These results allow us to answer ***RQ1***, ***RQ2***, and ***RQ3***. That is, developers using the feature location are able to complete the feature location tasks faster (***RQ1***), more accurately (***RQ2***), and with more confidence (***RQ3***).

Table 6: Statistics for control and experiment groups.

Measure	Group	Min	Max	Mean	SD
Time to complete (seconds)	***Control***	18.0	390.0	88.6	64.0
	Experiment	8.0	118.0	28.4	19.8
Confidence in location? (0-10)	***Control***	1.0	10.0	7.4	2.4
	Experiment	3.0	10.0	8.5	1.5
Location quality score? (0-100)	***Control***	0.0	100.0	76.4	39.2
	Experiment	0.0	100.0	88.4	24.6

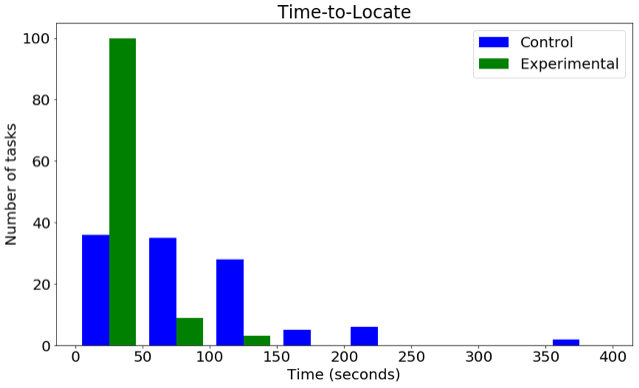

Figure 3: Time-to-locate result histogram.

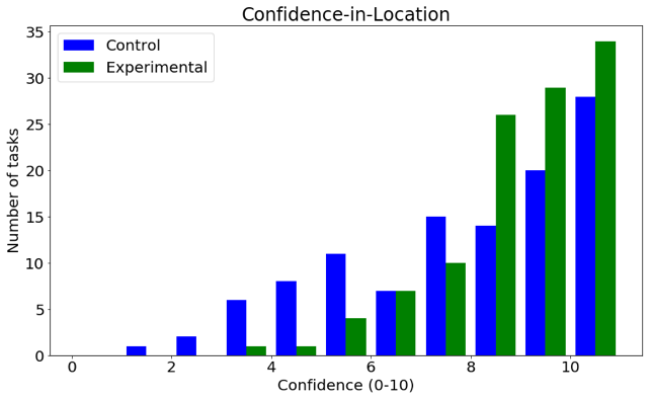

Figure 4: Confidence-in-location result histogram.

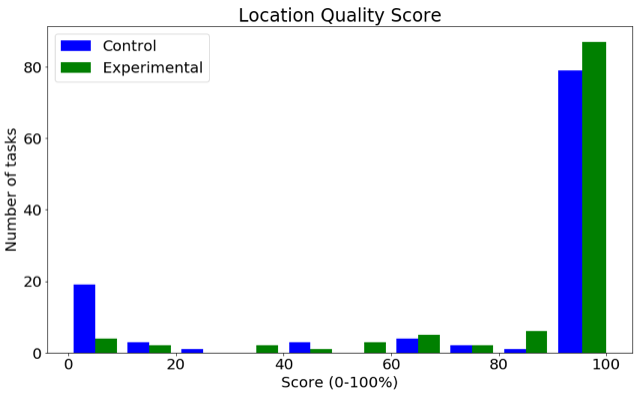

Figure 5: Location quality score result histogram.

Table 7: Experimental hypothesis testing with paired t-test

Measure	T-Test	P-Value	Cohen's d

	(paired)		
Time to complete	10.1	Reject (< 0.00001)	Very Large (-1.27)
Confidence in location	-5.5	Reject (< 0.00001)	Medium (0.54)
Location quality score	-3.6	Reject (0.0005)	Small (0.37)

Table 8: Experimental hypothesis testing with Wilcoxon signed-rank test.

Measure	Wilcoxon	P-Value
Time to complete	163.0	Reject (< 0.00001)
Confidence in location	471.5	Reject (< 0.00001)
Location quality score	379.5	Reject (0.00781)

Table 9: List of threats are identified in the left column. How each threat is mitigated is presented in the right column.

Threat / Risk	Mitigation
Software evaluated is not realistic	Use software in commercial production use, developed under test-driven-development practices.
Experimental software is not realistic	Design software after real-world interfaces, use common GUI affordances; verify realism with participants.
Feature location method is not useful in real word	Simulate a real world experience; evaluate usefulness with controlled experiment.
Participant pool is not realistic for real world	Draw participants solely from professional software developers, currently being paid to write software.
Skill factors – experience, training, familiarity with subject	Collect experimental and control sample tasks in pairs from every participant; normalize by participant if required.
Features differ in difficulty of location	Collect experimental and control sample tasks in pairs from every feature
Time of day,participant fatigue, etc.	Collect experimental and control sample tasks in pairs at nearly the same time in the same session.

With positive answers to the sub-research questions ***RQ1***-***RQ5***, we can answer the main research question. That is, the proposed approach of using iteration history to form invoking and excluding tests can be used effectively by developers for feature location tasks in real-world agile environments and projects.

5 LIMITATIONS AND THREATS

There are several concerns about the applicability of this method that might reduce its effectiveness. The purpose of the study is to evaluate the applicability of the approach despite these limitations. Table 9 contains a list of threats to validity and the measures taken to mitigate the threats.

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. In the worst case this approach will identify the code associated with both features. While not ideal, this will still significantly narrow down code.

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.

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 will not mask the later base feature, so that code will show up as part of the selected feature.

6 RELATED WORK

The task of feature location has long been studied via a variety of methods and techniques [13]. The most closely related work to what is presented here are approaches that use invoking and excluding tests. This paper presents the first advancement to such techniques in a number of years.

Wilde initially proposed software reconnaissance as a method to locate features in instrumented code [5]. 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 [14]. Subsequent work include White extending the technique from C/C++ to locating features in Ada [15], a comparisons study [16], extending the method to distributed systems [17], and testing the approach in a commercial environment [6]. Of note is Simmons remarks on the dependence of proper test selection [6]. Wong et al [7] discuss the problems with test selection.

Eisenberg tried to reduce the sensitivity due to test selection issues by introducing dynamic feature traces; using a test to feature set mapping to create a dynamic graph of "ranks" and "calls" [8]. Once feature set mapping to tests was complete, the trace analysis and dynamic feature trace 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.

Excluding tests were not used subtractively, but rather to provide computational comparison to the invoking tests. Further, the evaluation of the quality of the results is basically a self-assessment of "usefulness". Finally, Eisenberg describes the unsuitability of TDD tests for use with software reconnaissance because developers in agile methods don't generally write excluding tests. Eisenbarth uses concept analysis to find features in source code. He uses scenarios (a series of program use cases) to map features onto computational units of various granularities.

Korel and Laski proposed the concept of dynamic slicing to mark statements, array elements, and pointer variables that were used in the execution of a feature [18]. Note that this differs from code coverage in its inclusion of data elements in the profiling of a specific execution session.

Korel and Rilling proposed using dynamic slices for program understanding, including feature location [19]. The difference in content of two dynamic traces can be used in the same way as a comparison of execution traces. Hall proposed algorithms that could generate simultaneous dynamic slices from more than one intersecting test collection [20]. De Lucia noted that these slices could be used much the same as execution traces in Wilde's feature location methods [21]. Sim, Clarke, and Holt 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 [37].

Most recent work on feature location have utilized a variety of other techniques besides invoking/excluding tests to solve the problem. Marcus and Maletic introduced the use of the Information Retrieval (IR) technique Latent Semantic Indexing (LSI) for the problem of concept location [22]. Poshyvanyk combined LSI with dynamic execution trace methods [23]. More recent work includes [9][24] [25][26] [27][28] [29][30] [31][32] [33][34] [35][36].

7 CONCLUSION AND FUTURE WORK

We presented a model for feature location in an agile development environment. The model solves a significant drawback associated with other software reconnaissance based methods. Briefly, software reconnaissance uses the set difference of lines executed between invoking tests, tests that invoke a particular feature, and excluding tests, tests that are as similar as possible to the invoking tests, yet do not invoke the feature to perform feature location. The quality of the results returned by a software reconnaissance method is heavily dependent on the selection of quality invoking and excluding tests. 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.

By using inherent properties of agile development, the model allows for the automatic identification of invoking and excluding tests. As features are implemented in iterations, the tests added during an iteration to fully test a feature serve as invoking tests for the feature(s) developed. Since, tests must pass at the end of an iteration, no previous iteration will have tests that trigger the current iteration's features, and thus the set of all previous tests can serve as excluding tests.

A tool was developed implementing the model and the approach is validated in a comparison study with manual feature location. The results show, with significance, that the model allows for more accurate feature location identification in significantly less time and with greater developer confidence.

As there are a few limitations to the approach, we plan to investigate in the future the extent to which these limitations occur, as well as, a means of improving the approach to affectively deal with the limitations. In addition, we plan to apply the approach in an empirical comparison study on a significantly larger code base.

REFERENCES

[1] K. Bennet, V. Rajlich, and N. Wilde, "Software Evolution and the Staged Model of the Software Lifecycle," *Advances in Computers*, vol. 56, pp. 1–54, 2002.

[2] Keith H. Bennett and Vaclav Rajlich, "Software maintenance and evolution: a roadmap," presented at the International Conference on Software Engineering - The Future of Software Engineering Track, 2000, pp. 73–87.

[3] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 213–222.

[4] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 255–265.

[5] Norman Wilde, "Faster reuse and maintenance using software reconnaissance," *Technical Report SERC-TR-75F*, 1994.

[6] Sharon Simmons, Dennis Edwards, Norman Wilde, Josh Homan, and Michael Groble, "Industrial tools for the feature location problem: an exploratory study," *Journal of Software: Evolution and Process*, vol. 18, pp. 457–474, 2006.

[7] W Eric Wong, Swapna S Gokhale, Joseph R Horgan, and Kishor S Trivedi, "Locating program features using execution slices," presented at the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET'99), 1999, pp. 194–203.

[8] Andrew David Eisenberg and Kris De Volder, "Dynamic feature traces: Finding features in unfamiliar code," presented at the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 337–346.

[9] M. Chochlov, M. English, and J. Buckley, "A historical, textual analysis approach to feature location," *Information and Software Technology*, vol. 88, pp. 110–126, Aug. 2017.

[10] C. S. Corley, K. L. Kashuda, and N. A. Kraft, "Modeling changeset topics for feature location," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 71–80.

[11] K. Beck, *Extreme Programming Explained*. Addison Wesley, 2000.

[12] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, pp. 751–761, Oct. 1996.

[13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, pp. 53–95, 2013.

[14] N. Wilde and M. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Software Maintenance: Research and Practice*, vol. 7, pp. 49–62, 1995.

[15] Laura J White and Norman Wilde, "Dynamic analysis for locating product features in Ada code," *ACM SIGAda Ada Letters*, vol. 21, pp. 99–106, 2001.

[16] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds, "A comparison of methods for locating features in legacy software," *The Journal of Systems and Software*, vol. 65, pp. 105–114, 2003.

[17] Dennis Edwards, Sharon Simmons, and Norman Wilde, "An approach to feature location in distributed systems," *Journal of Systems and Software*, vol. 79, pp. 57–68, 2006.

[18] Bogdan Korel and Janusz Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155–163, 1988.

[19] Bogdan Korel and Jurgen Rilling, "Dynamic program slicing methods," *Information and Software Technology*, vol. 40, pp. 647–659, 1998.

[20] Robert J Hall, "Automatic extraction of executable program subsets by simultaneous dynamic program slicing," *Automated Software Engineering*, vol. 2, pp. 33–53, 1995.

[21] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro, "Understanding function behaviors through program slicing," presented at the Program Comprehension, 1996, Proceedings., Fourth Workshop on, 1996, pp. 9–18.

[22] Jonathan I. Maletic and Andrian Marcus, "Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding," presented at the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00), 2000, pp. 46–53.

[23] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, 2007.

[24] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *Empirical Software Engineering*, vol. 14, no. 1, pp. 93–130, Feb. 2009.

[25] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, Sep. 2010.

[26] G. Scanniello and A. Marcus, "Clustering Support for Static Concept Location in Source Code," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 1–10.

[27] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent Dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, Jun. 2014.

[28] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance," New York, NY, USA, 2007, pp. 14–23.

[29] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the Use of Deep Learning for Feature Location," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 556–560.

[30] N. Alhindawi, J. I. Maletic, N. Dragan, and M. L. Collard, "Improving Feature Location by Enhancing Source Code with Stereotypes," presented at the 29th IEEE International Conference on Software Maintenance (ICSM'13), 2013, pp. 1–10.

[31] Y. Zhang, D. Lo, X. Xia, T. D. B. Le, G. Scanniello, and J. Sun, "Inferring Links between Concerns and Methods with Multi-abstraction Vector Space Model," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 110–121.

[32] G. Scanniello, A. Marcus, and D. Pascale, "Link analysis algorithms for static concept location: an empirical assessment," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1666–1720, Dec. 2015.

[33] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 151–160.

[34] X. Peng, Z. Xing, S. Pan, W. Qian, V. Rajlich, and W. Zhao, "Reflective feature location: knowledge in mind meets information in system," *Sci. China Inf. Sci.*, vol. 60, no. 7, p. 072102, Jul. 2017.

[35] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation," in *2008 15th Working Conference on Reverse Engineering*, 2008, pp. 155–164.

[36] M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," in *2010 IEEE 18th International Conference on Program Comprehension*, 2010, pp. 14–23.

[37] Susan Elliott Sim, Charles L. A. Clarke, and Richard C. Holt, "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers," presented at the 6th International Workshop on Program Comprehension (IWPC'98), 1998, pp. 180–189.