Exploring the Neighborhood with Dora to Expedite Software Maintenance *

Emily Hill, Lori Pollock, and K. Vijay-Shanker
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{hill, pollock, vijay}@cis.udel.edu

ABSTRACT

Completing software maintenance and evolution tasks for today's large, complex software systems can be difficult, often requiring considerable time to understand the system well enough to make correct changes. Despite evidence that successful programmers use program structure *as well as* identifier names to explore software, most existing program exploration techniques use either structural *or* lexical identifier information. By using only one type of information, automated tools ignore valuable clues about a developer's intentions—clues critical to the human program comprehension process. In this paper, we present and evaluate a technique that exploits *both* program structure and lexical information to help programmers more effectively explore programs. Our approach uses structural information to focus automated program exploration and lexical information to prune irrelevant structure edges from consideration. For the important program exploration step of expanding from a seed, our experimental results demonstrate that an integrated lexical- and structural-based approach is significantly more effective than a state-of-the-art structural program exploration technique.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids*; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*reverse engineering*

General Terms: Human Factors, Reliability

Keywords: Natural language program analysis, program exploration, software maintenance, software tools

1. INTRODUCTION

Completing a software maintenance or evolution task first requires understanding the existing software, followed by performing the actual modification and then revalidating the subsequent software version [6]. For maintenance tasks such as debugging, the first step of finding and understanding the code relevant to a bug fix tends to take more time than actually fixing the bug [20, 21]. With as much as 60-90% of software life cycle resources spent on program maintenance [6, 13], there is a critical need for automated tools to help explore and understand today's large and complex software.

Automated tools can draw from the variety of information developers use to explore programs, such as expert knowledge, external documentation, or dynamic information. Regardless of additional available information, the developer will always be interested in the source code itself. Thus, in this paper we focus on static information available from the source code, in the form of programmer-defined structure and lexical identifier names. Most automatic exploration techniques that leverage this structural and lexical information focus on either search or navigation.

When searching code, maintainers typically use *lexical* information found in the program's comments and identifiers. Identifier names often communicate a programmer's intent in writing pieces of code [8, 22], and therefore provide valuable information to the developer. To explore a program by searching, the developer enters a query related to the maintenance task and the search tool retrieves potentially relevant program elements. Thus, searches find disparate program elements relevant to a query, but ignore structural information that may help eliminate irrelevant results.

When navigating code, maintainers often explore based on the program *structure* by following method call chains or control flow, as well as dependence, variable def-use, or type hierarchy relationships. In contrast to lexical searches, navigation suggestions utilize program structure information to automatically recommend relevant sections of code [32, 39]. Unfortunately, a single program element may be structurally connected to tens or hundreds of other elements, when only a handful are relevant to the maintenance task.

Thus, most existing program exploration techniques use either lexical *or* structural information, despite evidence that successful programmers use lexical *as well as* structural information to explore programs [33]. By using only one kind of information, automated tools ignore valuable clues about a developer's intentions [5]—clues critical to the human program comprehension process. *By utilizing lexical* as well as *structural program information, we can create automatic program exploration tools that mirror how humans attempt to understand code.*

*This material is based upon work supported under an NSF Graduate Research Fellowship and CCF-0702401.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

In this paper, we present and evaluate an automated approach for focusing program exploration that exploits *both* structural and lexical information. We have implemented our technique as an automated tool, **Dora the Program Explorer**.[1] Dora takes as input a natural language query related to the maintenance task and a program structure representation to be explored. Dora then outputs a subset of the program structure relevant to the query, called a *relevant neighborhood*. In this paper, we assume a starting point, or seed, in the program structure representation and we focus on finding the relevant neighborhood for this seed. The current implementation of Dora uses the program call graph [14] as the program structure representation to be explored, and methods as the seed elements.

This paper makes the following contributions:

- A technique to automatically identify the relevant neighborhood of a call graph using lexical information, including a technique to score method relevance with respect to a natural language query
- An Eclipse plug-in, Dora, that implements our technique and enables programmers to visualize a relevant neighborhood for software maintenance tasks
- Quantitative evaluation against an existing structural-based technique and two naive lexical- and structural-based techniques for a baseline comparison

Our experimental results demonstrate that an integrated lexical- and structural-based approach is significantly more effective than a state-of-the-art structural program exploration technique.

2. IMPROVING THE STATE-OF-THE-ART

Developers use a variety of approaches to explore programs and find code relevant to maintenance tasks. In this section, we outline the most relevant approaches.

Navigation-based Exploration Approaches

Navigation-based program exploration techniques help developers navigate structural dependencies. Unfortunately, most existing tools require the developer to initiate every exploration step and manually select every structure edge to be expanded [9, 34, 38]. In contrast, Dora automatically explores highly relevant structure edges to save the developer time and effort by producing a *relevant neighborhood*.

The most closely related exploration technique is Robillard's structural topology approach, Suade [32, 43], which automatically generates suggestions for program investigation based on a seed method set, or concern. The Suade approach uses structural relations between program elements (calling a method, being called by a method, accessing a field, and being accessed by a method) to evaluate the *specificity* and *reinforcement* that a given element has upon another. Elements that have fewer structural relationships are considered more specific, and are therefore given a higher relevance score. Reinforcement increases the score of elements that have more structural connections to elements already in the concern. Our approach differs from Suade in that we use lexical and structural information to explore a program, rather than just structure. Because both techniques use different information to score relevant program elements, it may be possible to create a hybrid approach that combines both techniques.

Another technique that automatically includes relevant structural edges is program slicing. Slicing techniques use program dependence relationships such as control and data dependence to extract the parts of a program that may affect a point of interest [39, 41, 44]. The slices of the program can be used to aid program comprehension. Unfortunately, slices tend to be large and can be expensive to calculate. Reducing expense has the trade-off of making slices more conservative and therefore containing even more irrelevant information. To potentially reduce cost and improve relevance, our lexical scoring mechanism could be applied to slicing techniques as an additional stopping criteria.

Search-based Exploration Approaches

Most search-based approaches to program exploration use the lexical information in comments and identifiers to locate methods relevant to a maintenance task. There are regular expression-based searching techniques such as UNIX `grep`, however, most current research focuses on modern information retrieval (IR) [24, 31] and natural language [37] searching techniques. To make queries more effective at locating relevant code, approaches have been suggested to help find the concept words in software [27, 37]. These approaches are complementary to our work to potentially discover a seed method set, which is used as input to our current approach.

Other search-based approaches automatically link documentation to source code by using lexical information [2, 23] or a combination of lexical and structural information [45]. Although these approaches are fully automatic, they require accurate documentation in addition to meaningful identifier names, whereas our approach needs only meaningful identifier names. However, incomplete, nonexistent, or inaccurate documentation could hinder the searching effectiveness of these approaches. In addition, the documentation may be written at a coarser granularity than the maintenance task being completed. With no mechanism to retrieve finer granularity matches, the developer is forced to search for only those features that are at the documentation granularity. By leveraging the identifiers of a program and allowing user-specified queries, Dora is capable of searching at multiple levels of granularity.

Some search-based techniques utilize structure information, rather than lexical information. JQuery [19] combines structural queries with navigation and integrates multiple kinds of program structure relationships in a contextual view. Strathcona [15] uses the structure of the code that a developer is working on to automatically recommend relevant example code from a repository. These approaches are complementary to our work to potentially discover a seed method set.

We are aware of one other search technique that combines lexical as well as structural information. Sourcerer [3] uses keyword queries in addition to program structure to find relevant examples from a large repository of open source projects. Although the work focuses on a different problem, the approach appears very related. Unfortunately, there are not enough details in the limited initial publication to fully compare the approach to Dora.

[1] Dora comes from *exploradora*, the Spanish word for a female explorer.

Software Architecture Recovery

Developers also can use architecture recovery techniques to understand a system [5, 26, 28]. During architecture recovery, developers use lexical patterns in conjunction with source structure models to locate high-level concepts in code. The developer is responsible for articulating the key concepts of the maintenance task in a regular expression query that may also require specifications as to which code structures should be searched [26, 28]. In contrast, Dora takes as input a simple natural-language-based query. By using natural language query terms rather than regular expressions, Dora can utilize more advanced information retrieval techniques such as stemming [29] and tf-idf [42].

Program Structure Visualization

A number of approaches have been suggested for whole-program visualization and navigation [4, 7, 12, 18, 40]. Some approaches have made efforts to restrict the information presented so as not to overwhelm the developer by adding zooming or fisheye viewing capabilities [7, 18, 40] or aggregating dependencies to a higher level [4]. By using Dora, we believe that these tools can be further improved by focusing the developer's attention on information that is likely to be relevant to a maintenance task.

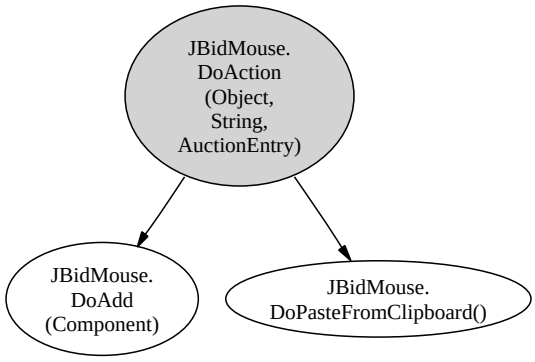
Program Structure Representations

A variety of structural program models have been proposed to enhance program comprehension and facilitate maintenance tasks, such as system dependence graphs [16], program slices [41, 44], type hierarchies [11] and call graphs [14]. Although our general strategy can be applied to any type of structural program model, we focus this paper on call graphs as an intuitive model that allows developers to quickly comprehend interactions between large sections of code. Call graphs are relatively inexpensive to calculate, and have the advantage of representing even scattered code well because call graphs are indifferent to class decompositions.

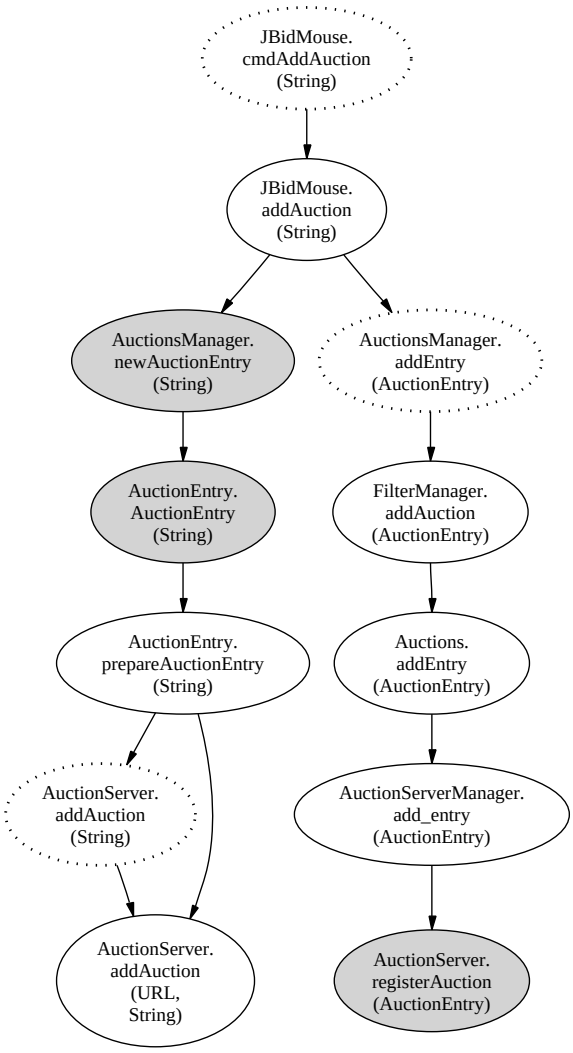
3. COMBINING LEXICAL & STRUCTURAL INFORMATION: AN EXAMPLE

In this section, we present an example motivating why utilizing both lexical and structural information is important for automated program exploration tools. Consider the 'add auction' concern[2] in the open source auction sniping program, jBidWatcher.[3] Figure 1 is the subgraph of the jBidWatcher call graph relevant to the concern. The shaded methods are those missed by a simple lexical search on the query 'add*auction.'

The concern consists of two connected components: the trigger, shown in Figure 1(a), and the handler, shown in Figure 1(b). The trigger code handles the user-initiated GUI event, which adds the 'add auction' event request to the program's application event queue. The handler code processes the 'add auction' event, removing it from the event queue and adding the auction to the program's internal data structures. Within each component, the methods are statically connected by calling relationships. The two components are linked by a data dependence on the event queue.



(a) Trigger connected component.



(b) Handler connected component.

Figure 1: 'Add Auction' concern.

[2]A *concern* is a high-level idea, or feature, implemented in code.

[3]`sourceforge.net/projects/jbidwatcher` version 1.0pre6

```
1  private void DoAdd(Component src) {
2    String endResult;
3    String prompt =
4      "Enter the auction number to add";
5    endResult = promptString(src, prompt, "Adding");
6
7    // They closed the window or cancelled.
8    if (endResult == null) return;
9    MQFactory.getConcrete("user")
10     .enqueue(ADD_AUCTION + endResult);
11 }
12
13 private void DoPasteFromClipboard() {
14   String auctionId = getClipboardString();
15   // ...
16
17   if(auctionId != null) {
18     MQFactory.getConcrete("user")
19           .enqueue(ADD_AUCTION + auctionId);
20   }
21 }
```

Figure 2: Triggering an 'add auction' event. It is not obvious from the method name that doPasteFromClipboard is relevant to adding an auction.

Combining lexical and structural information for automated exploration tools is helpful for a number of reasons. Through this example, we highlight the main reasons:

Automatically eliminate breadth choices. Finding the methods that handle the 'add auction' event by looking at the callees of DoAction is no trivial task. DoAction calls 38 methods within the same class that handle various user actions such as bidding, searching, or adding an auction. For the 'add auction' task, only two of the 38 methods are relevant: doAdd and doPasteFromClipboard, shown in Figure 2. Although method signatures are usually a good indication of relevance, in this example, a developer skimming the callees might miss the relevant method doPasteFromClipboard. Thus it is important for lexical techniques to analyze the signature as well as the source code of each method to include non-obvious relevant methods while still eliminating irrelevant call edges.

Automatically expand deep call chains. Looking at the handler component in Figure 1(b), there are 12 relevant methods and two call chains that are 7 methods deep. The few unrelated callers into the call chain are easy to eliminate using lexical name information.

Find relevant elements missed by lexical searches. The most effective simple search for this concern is the regular expression 'add*auction'. Using Eclipse's [17] simple lexical search on methods rather than files, the query matches 50 methods, 11 of which are relevant. The 3 methods in the top ten results (ranked by number of matches) have dashed lines in Figure 1. The lexical search misses 4 relevant methods that are easily found using structural call edges.

Recursive exploration. Lastly, the jBidWatcher example in Figure 1 illustrates how using lexical information to prune irrelevant call edges enables recursive exploration. Most existing techniques for call graph navigation either display the entire call graph [7, 12] or explore a single edge at a time [38]. Another navigation technique is Eclipse's frequently-used call hierarchy feature [17], which allows developers to recursively view *either* all descendants *or* all ancestors of a method. These approaches lack the ability to recursively recommend relevant callers *and* callees. This capability would allow developers to view sibling calling relationships in addition to ancestor and descendant relationships.

For example, starting from the Auctions.addEntry method, it is impossible to realize that Auctions.addEntry and AuctionEntry.AuctionEntry share the caller JBidMouse.addAuction in Eclipse's call hierarchy without changing to a different starting point. However, by using lexical information to prune irrelevant edges, exploration tools can automatically discover relationships such as shared callers with no additional user intervention.

In summary, combining both lexical and structural information enables exploration tools to automatically prune irrelevant structural edges. By eliminating irrelevant edges, exploration tools can recursively search a structural program representation to provide the maintainer with a broad, high level view of the code relevant to a maintenance task—without including the entire program.

4. PROGRAM EXPLORATION WITH DORA

Before detailing how Dora combines lexical and structural information to explore programs, we illustrate how developers can use Dora during maintenance. There are a number of maintenance situations where exploring with Dora is useful, such as: finding all the dependencies for a reusable code component, finding existing relationships to add a feature, analyzing dependencies to judge change effect propagation, or to better understand the flow of a concern. The following steps illustrate how developers can use Dora to find the relevant neighborhood of a maintenance task.

Step 1. Determine the query. The user should formulate a query related to the maintenance task. The query terms can come from simple lexical searches, expert recommendations, interactive query expansion [30, 37], or be derived directly from a maintenance request. If multiple terms are used in the program to refer to the same concept, all these terms should be included in the query. Since Dora is most effective when the query terms match the actual terms used in the code, users unfamiliar with the code or unsure of the program terms used to refer to a particular concept in the code should plan to augment their query term selection.

Step 2. Identify the seed method set. There are a number of ways a maintainer can quickly locate a seed set. The seed set could come from expert recommendations, the developer's own prior knowledge, or by using search tools such as Eclipse's lexical search [17], Google Eclipse Search [31], or FindConcept [37].

Our exploration heuristic is designed to operate on static program models, which may be inadequate to represent all the dynamic relationships found in code. Therefore, for our approach to fully explore the code relevant to a maintenance task, the user must add seed methods that are not connected by statically available edges. It should be noted that only one method need be selected from each relevant connected component—our heuristic is designed to find the rest.

Step 3. Identify the relevant neighborhood. Our current Dora strategy follows call graph edges from the seed set, scoring each method's relevance to the query. The details of our relevance score are described in Section 5.

Step 4. Output the relevant neighborhood. Dora automatically displays the relevant call graph neighborhood to the user. In addition, the relevant methods are output to

a concern representation [36] for further investigation and future viewing. In reaction to the output, the maintainer can inspect an overall view of the relevant code and delve into the details of the most promising methods. If the user believes the results to be incomplete, the user can optionally add additional query terms or seed methods and rerun Dora.

5. AUTOMATICALLY IDENTIFYING THE RELEVANT NEIGHBORHOOD

The main contribution of Dora is the automatic identification of the relevant neighborhood by combining lexical and structural information. Starting from a seed method m in the seed set, Dora uses structural information by traversing structural call edges to find the set of callers and callees for m. The set of callers and callees become candidates for the relevant neighborhood. Next, Dora uses lexical information by scoring each candidates' relevance to the query, which we call the **method relevance score**. Candidates scored higher than a given threshold, t_1, are added to the relevant neighborhood. Candidates scored less than t_1 but more than a threshold t_2 are further explored to ensure they are not connected to more relevant methods. This use of two thresholds guards against missing very relevant methods that are connected by a borderline relevant method. Both thresholds are given a default value of $t_1 = 0.5$ and $t_2 = 0.3$, but are user specifiable. Finally, this exploration process is recursively repeated for each starting seed method and for each method added to the relevant neighborhood. In the remainder of this section we describe in detail how Dora uses lexical information to calculate the method relevance score.

5.1 Components of the Method Relevance Score

5.1.1 Term Frequency

The principal component of our method relevance score is how frequently query terms appear in a method, also known as *term frequency*. Term frequency (tf) is often used to determine document relevance in information retrieval (IR) [42]. The intuition is that the more frequently a word occurs, the more relevant the document, or method, is to the query. For example, in an addAuction method, the word 'auction' appears 25 times. In contrast, the word 'sort', a term irrelevant to adding an auction, appears only once.

The drawback of term frequency is that uninformative terms appearing throughout the program can distract from less frequent, but relevant, terms. Intuitively, the more methods that include a term, the less a term discriminates between methods. To address this issue, the IR community commonly multiplies a term frequency by its inverse document frequency (idf), called a *tf-idf* score [42]. The idf for a term t is calculated by dividing the total number of methods in a program by the number of methods that contain t, and taking the resulting number's natural log.

For example, consider the query 'add auction' from the auction sniping program jBidWatcher. Because the domain of jBidWatcher involves online auctions, the term 'auction' appears in 470 of the 1,812 methods in the program. In contrast, the word 'add' appears in only 261 methods. Therefore, occurrences of the term 'add' are given a higher *tf-idf* score than occurrences of 'auction'. Thus, more occurrences of the word 'auction' are required to get a *tf-idf* score as high as the less-used term 'add'.

Before counting the frequency of terms, we apply a simple preprocessing step to the query and methods. First, all the identifiers are split into terms based on non-alphabetic characters and camel case, similar to previous textual source analysis approaches [2, 23]. For example, *addAuction* and *add_auction* both become the terms 'add' and 'auction'. Next, the terms are converted into lower case and stemmed using Porter's stemming algorithm [29]. Stemming ensures that similar terms like 'auction' and 'auctioned' map to the same conceptual term of 'auction.' To count how frequently query terms appear in a method, we use a sum of the *tf-idf* scores for each query term appearing in the method.[4]

5.1.2 Method Features

In addition to how frequently a term occurs, our score takes into account *where* the query terms appear in the method. We consider the method name to be the most important indicator of relevance. Because method names have higher visibility in a program than, say, local variables, programmers typically select very descriptive method names [22]. We chose not to include other method signature information, such as the declaring class or package name, for the same reason that we use idf: classes and package names are shared by many system components and therefore are less able to differentiate between program elements.

As demonstrated by our example in Section 3, occasionally a method name alone does *not* indicate relevance to a maintenance task. Thus, we count the number of method statements containing a query term, multiplied by the term's idf. These *tf-idf* scores are summed and then normalized by the method length. Because longer methods are more likely to contain more query term occurrences, dividing by the method length ensures our scores are not biased to longer methods.

In addition to where terms are located, we also take into consideration whether a method is binary, i.e., whether it is a library method with no source code present. Although library methods are rarely explored by developers, highly relevant library method calls can provide additional information to relevant non-library methods. Therefore, we include highly relevant calls to library methods in the relevant neighborhood for context purposes. Thus, the *method features* that we consider for our scoring technique are *name*, *statement*, and *binary*.

5.2 Calculating the Method Relevance Score

Although we had some intuition into the features of a method that would be useful in determining relevance to the query, we were unsure how to weight them for the relevance score. To determine our weights, we applied logistic regression on a training set of methods. Logistic regression is a statistical technique to find the best fitting model for a binary dependent variable [1]. Given a set of features, or independent variables, logistic regression outputs a set of feature weights β and an intercept value α that best predict the training data. Unlike linear regression methods, which output a linear model, logistic regression outputs an exponential model. Therefore, given a feature vector x of size k, weights β, and intercept α, we apply the following equation

[4]In IR, cosine similarity is commonly used to determine the similarity score between a query and a document. However, in our experience we have found that cosine similarity scores do not translate well for programs, and thus use only tf-idf.

to calculate the probability p that a method is relevant to the query:

$$p = \frac{e^{\alpha+\beta_1 x_1+\ldots+\beta_k x_k}}{1+e^{\alpha+\beta_1 x_1+\ldots+\beta_k x_k}}$$

This equation will always give a value between 0 and 1, making it ideal for calculating probabilities.

For our training set, we used methods from nine concerns used in a previous concern location tool evaluation [37]. We included the methods in the concerns plus all methods one call edge away from any method in a concern. We manually inspected each method and annotated them as either relevant or irrelevant. Although the manual annotations are necessarily subjective, we have tried to limit bias by combining the input of three Java programmers.

After training the model, we define our method score (p):

$$p = \frac{e^{-0.5+-2.5*bin+name+0.5*statement}}{1+e^{-0.5+-2.5*bin+name+0.5*statement}}$$

In training our model, we considered other possible method features such as comments. However, none of the other features we tried with the model added to its predictive power. We would have liked to consider other types of methods in addition to *binary*, such as public, private, abstract, etc., but had insufficient training data for these categories. Therefore, we focused the model on the simplest variables that best predicted relevance: *binary*, *name*, and *statement*.

One might be tempted to blindly apply the scoring mechanism presented to search the whole program for relevant sections of code. However, our additional requirement that all elements included in the neighborhood be structurally connected to a seed method significantly restricts our search of the program to very relevant code. A whole-program scoring technique would need to be more sophisticated to filter out spurious occurrences of related words in contexts unrelated to the query.

6. EXPERIMENTAL EVALUATION

The purpose of our evaluation is two-fold:

1. To compare our integrated lexical- and structural-based approach against a state-of-the-art structural-based approach
2. To demonstrate that our sophisticated lexical scoring technique is an improvement over naive lexical scoring techniques in identifying the relevant neighborhood

6.1 Experiment Design

6.1.1 Variables and Measures

The independent variable in our study is the method scoring technique. The structural-based approach we compare against is Robillard's structural topology approach, *Suade* [32, 43]. *Suade* uses the *specificity* and *reinforcement* that a given method has upon another to recommend structurally relevant methods. Because we are evaluating the technique on singleton method sets, only the specificity of a program element is taken into account.

We include two additional lexical- and structural-based techniques in our study: boolean-AND (AND) and boolean-OR (OR). These techniques are baselines used to evaluate Dora's more sophisticated relevance score. These techniques output either 0 or 1: AND outputs 1 if *all* query terms appear in the method; OR outputs 1 if *any* query term appears in the method.

Crn.	Query	Prog.	$\cup$	$\cap_3$	$\cap_2$	$\frac{\cap_2}{\cup}$
C3	Task progress complete	Gantt	32	2	12	38%
C9	Update auction	JBid.	22	3	11	50%
C10	Download thumbnail image	JBid.	19	6	11	58%
C11	Execute auction bid	JBid.	13	2	7	54%
C12	Delete auction	JBid.	24	6	11	46%
C13	Toggle fold node	Free.	21	6	14	67%
C14	Zoom in out	Free.	25	2	14	56%
C16	Auto save file	Free.	6	2	5	83%

Table 1: Concerns and queries used in evaluation, in terms of number of methods.

Program	Version	NCLOC	# Classes	# Methods
Gantt	2.0.2	43,246	555	3,991
JBidWatcher	1.0pre6	22,997	183	1,812
Freemind	0.8.0	70,341	617	5,388

Table 2: Program Characteristics for concerns used in the evaluation.

The dependent variable in our study is the effectiveness of each technique, measured in terms of the commonly used IR measures, precision and recall. *Precision* is the fraction of methods reported by the technique that are relevant, calculated by dividing the number of relevant methods reported by the total number of methods reported. *Recall* is the fraction of relevant methods reported, calculated by dividing the number of relevant methods reported by the total number of actual relevant methods. High precision implies a scoring technique returns few irrelevant methods, whereas high recall implies the scoring technique misses few relevant methods. Since ideal techniques have both high recall and high precision, the F measure is commonly used to combine both precision and recall into a single measure. The *F measure* is defined as the harmonic mean of precision and recall, and is high only when both precision and recall are high. Thus, a high F value can be interpreted as the best possible combination of precision and recall. We use the F measure to evaluate the performance of each scoring technique.

6.1.2 Subjects

To evaluate each technique, we use sets of methods from concerns as seeds. Thus, each subject in our study is a <concern, query> pair. To avoid investigator bias in determining our own concerns for evaluation, we selected 8 concerns from a recent study of the concept assignment problem on 4 open source Java programs [35]. Methods were selected for each concern by 3 independent developers, with varying levels of agreement (overlap in methods selected). Details of the concerns are shown in Table 1. The column labeled $\cup$ is the union of methods selected by all three developers in the study, $\cap_3$ is the number agreed upon by all three developers, $\cap_2$ is the number at least two agreed upon, and the last column shows the percent developer agreement.

Because conflicts of agreement could indicate poor quality method sets for a concern, we used only those concerns where at least two developers agreed on 35% or more of the relevant methods ($\frac{\cap_2}{\cup}$ in Table 1). We considered any method selected by at least two developers to be relevant to a concern. To put the concerns into context, the program characteristics for the concerns are presented in Table 2.[5]

[5]Calculated using the Eclipse Metrics Plug-in, `metrics.sourceforge.net`.

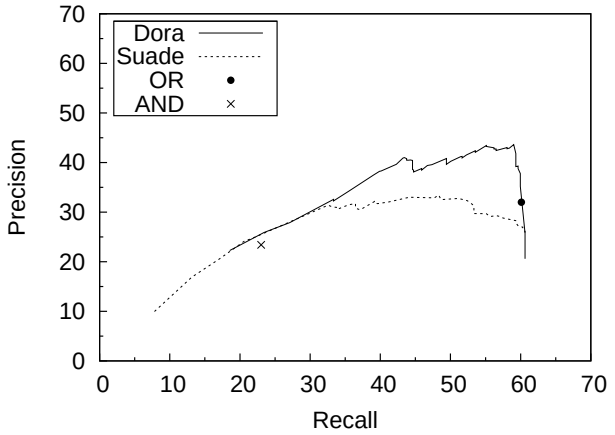


Figure 3: Precision-Recall Graph. *Suade* and *Dora* were evaluated at various thresholds ranging from 0 to 1 (AND and OR require no threshold). Each point represents precision and recall averaged over a given threshold, with decreasing threshold values from left to right.

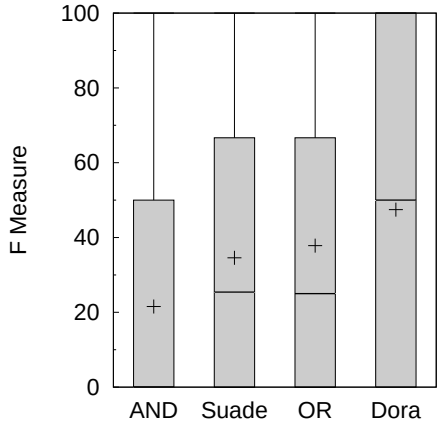


Figure 4: F Measure across techniques. The shaded box represents 50% of the data, from the 25th to 75th percentiles, the horizontal bar represents the median, and the plus represents the mean. *Dora* performs significantly better than *Suade* ($\alpha = 0.05$).

The one component of our subject <concern, query> input missing from these concerns is the query. To avoid investigator bias, the queries were chosen by an independent researcher involved in the concept assignment study who had no knowledge of our scoring technique. The queries were selected by looking at the concern descriptions, lexical searches, and a query expansion mechanism [37].

Thus, we evaluated each scoring technique on 8 <concern, query> pairs, containing a total of 160 seed methods. The median number of caller and callee edges per seed is 7, with a minimum of one edge and a maximum of 62. A total of 1885 call edges were evaluated by each technique, with overlap.

6.1.3 Methodology

Although the ideal evaluation would recursively explore from each seed method, incorrect identifications would propagate and unfairly skew the results. Therefore, our evaluation compares the effectiveness of each technique applied *one edge away* from a single seed method. For each method m in the set of evaluation concerns, we applied each scoring technique to all the callers and callees of m, and calculated the precision and recall for m. We used Eclipse [17] to generate the structural information used by the techniques.

Both *Suade* and *Dora* output relevance scores ranging from 0 to 1. However, to evaluate the techniques in terms of precision and recall requires a threshold to map these scores to 0 *or* 1. Because this threshold selection is a potential threat to validity, we evaluated the threshold performance on the training data set used for Dora. We evaluated the precision and recall at threshold levels varying from 0 to 1 at 0.005 intervals, and selected the threshold for each technique that maximized the mean F measure. We found that *Suade* performed best at a 0.3 threshold, and *Dora* at a threshold of $t_1 = 0.5$.[6] Based on these thresholds, the results for *Dora* and *Suade* were partitioned into relevant (1) and irrelevant (0) scores before calculating precision and recall.

[6]Because we are only scoring methods one edge from a seed, Dora's threshold t_2 is unnecessary.

6.2 Results

Overall, we found *Dora* to be the most successful technique, and structural-based *Suade* to be competitive with the naive lexical- and structural-based OR. Of all the techniques, naive AND had the worst performance.

6.2.1 Precision, Recall, and Threshold Variation

Figure 3 shows a Precision-Recall graph for all four techniques. *Suade* and *Dora* were evaluated at thresholds ranging from 0 to 1 at 0.005 intervals (AND and OR require no threshold). Each point represents precision and recall averaged over a given threshold, with decreasing threshold values from left to right. (A high threshold means fewer methods are scored as relevant, and therefore recall is low.)

For both *Dora* and *Suade*, the recall progressively increases as the threshold decreases, whereas the precision increases with the threshold and drops off again when the threshold gets too high. The peak in *Dora*'s F measure performance at $(R = 59, P = 43)$ occurs at the 0.5 threshold. The peak in *Suade*'s performance at $(48, 33)$ occurs at the 0.41 threshold. At $(23, 23)$, AND performs similarly to *Dora* and *Suade* at higher than optimal thresholds, 0.99 and 0.64, respectively. At $(60, 32)$, OR performs similarly to *Dora* and *Suade*, with less than peak precision for *Dora*, and slightly better than peak recall for *Suade*. Figure 3 demonstrates how *Dora* outperforms *Suade* in terms of precision over many thresholds, and performs equivalently in terms of recall. *Specifically, Dora outperforms Suade in precision for thresholds* 0.05–0.98 *(Dora) and* 0.0–0.59 *(Suade).* *Suade* and *Dora* perform equivalently on the training set for all other thresholds.

6.2.2 Overall Performance

Figure 4 shows the overall performance of each technique. Each bar shows the distribution of F measures calculated for each seed method across all the concerns. The shaded box represents 50% of the data, from the 25th to 75th percentiles. The horizontal bar represents the median, and the plus represents the mean.

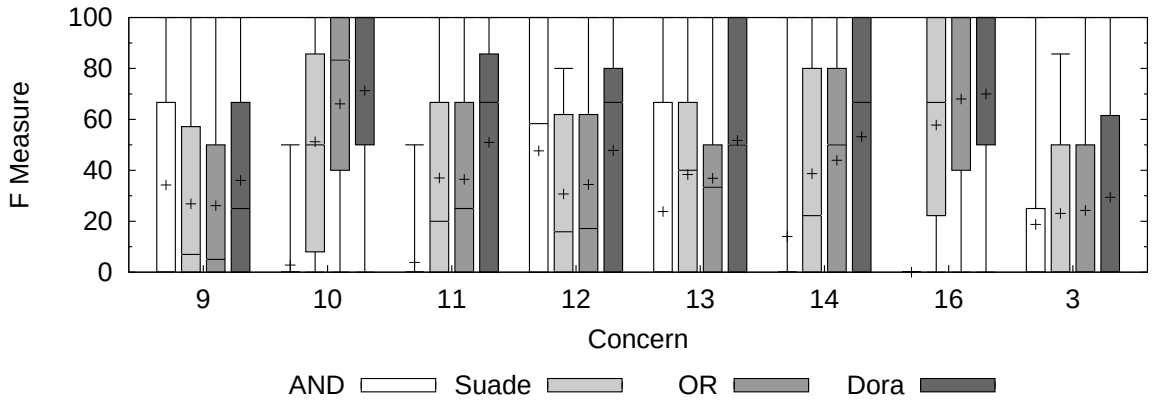


Figure 5: F Measure for each concern by technique. The trends illustrated in Figure 4 are also seen for most of the <concern, query> pairs.

Since each shaded box extends from 0, at least 25% of the 160 methods considered by each technique have 0% recall and precision. *However, Dora achieves 100% precision and recall for 25% of the data—more than any other technique. Suade* and *OR* appear to perform similar to one another, although *OR* has a slightly higher mean F measure.

To confirm these observations, we performed a Bonferroni mean separation test[7] at $\alpha = 0.05$. *Dora* performs significantly better than structural-based *Suade*, with a mean difference of 12.9, although neither *Dora* nor *Suade* are significantly different from *OR* (with mean differences of 9.6 and 3.3, respectively). All the approaches outperform *AND* with statistical significance.

6.2.3 Performance by Concern

Figure 5 shows the performance for each technique, separately for each concern. Although each concern does not contain sufficient data to judge significant differences, the trends in Figure 5 follow the same pattern as the overall performance results in Figure 4, with $AND < Suade \leq OR < Dora$. Concerns 9 and 12 are an exception to the general trend, with *AND* outperforming most techniques. For both of these concerns, *AND* had much higher precision than *OR* and *Suade*. This was due to the fact that most of the relevant methods contained both query terms. Because *AND* performed so poorly for the remainder of the concerns, *AND* is clearly very sensitive to the query.

6.3 Threats to Validity

Because the selection of methods relevant to a concern is a necessarily subjective task, the correctness of the concerns used in the evaluation could be a threat to validity. The developers were unfamiliar with the programs and tasks assigned to them, and were advised to spend less than an hour selecting program elements relevant to the concerns. For this reason it is possible that the developers missed relevant methods or even included irrelevant ones, which could impact the precision and recall of the techniques evaluated in the experiment. However, we do not believe that the developers consistently selected methods incorrectly, and we tried to alleviate this threat by only using concerns with at least 35% agreement. Since all the techniques are subject to the same vulnerability, we do not feel this is a serious threat to the validity of the study.

Four of the evaluation concerns were from a program that was used during training, although with three *different* concerns. This could affect our results by potentially giving Dora an unfair advantage. However, a T-Test revealed that for each technique, there was no significant difference[8] in mean F measures between concerns from completely new programs, and those concerns of a program used by three out of nine training concerns.

We minimized threats to conclusion validity by carefully selecting a threshold for *Dora* and *Suade*. We evaluated precision and recall at various threshold levels on the same training set, and selected the optimal threshold for each technique to maximize the mean F measure. In addition, we have shown a precision-recall graph (Figure 3) that illustrates our conclusions independent of threshold selection.

A second threat to conclusion validity is the application of *Suade* to singleton method sets, rather than partial concerns with multiple methods. Restricting the input to singleton method sets forces *Suade* to rely solely on its specificity scoring component, and perhaps to perform suboptimally. Because previous *Suade* evaluation [32] used such singleton method sets, we do not feel this is a serious threat to the conclusions of our study.

We minimized internal threats to validity by having each scoring technique operate on the same structure data. We minimized the potential for investigator bias by using concerns and queries selected by software engineers outside the investigators.

Because we evaluated the techniques on 8 concerns from 3 open source Java programs, the results of this study may not generalize to all programs and <concern, query> combinations. In addition, any developer naming conventions could have affected the lexical-based approaches.

[7]Before performing the contrasts, we applied the ANOVA F-Test to test that the means were significantly different ($p < 0.0001$). Next, we performed the Bonferroni mean separation test, which evaluates multiple mean comparisons and controls the experimentwise error rate. The error rate is controlled by adjusting the α for each pairwise comparison such that the experimentwise error rate of $\alpha = 0.05$ is preserved.

[8]At the 5% significance level: *AND* $p = 0.14$, *OR* $p = 0.53$, *Suade* $p = 0.77$, *Dora* $p = 0.39$.

6.4 Discussion

In this study, we found that the integrated *Dora* approach outperformed the purely structural *Suade* approach, motivating further development of integrated structural and lexical techniques.

In addition, we found that not all integrated approaches outperformed a purely structural approach. For example, the naive lexical scoring mechanism (OR) performed equivalently to structural-based *Suade*, and AND performed worse. Because AND requires all query terms to be present, it is very sensitive to the selected query terms. Thus, the success of a lexical- and structural-based technique is highly dependent on the performance of the lexical scoring technique.

Of the three lexical scoring techniques, *Dora* outperformed OR and AND. In Figure 4, *Dora* clearly outperforms OR and AND, although not with statistical significance over OR. However, Figure 5 shows that *Dora* performs as good as or better than OR in every concern. Further, there are additional lexical method features that could enhance *Dora* in the future, which might offer significant improvements over the naive OR technique.

In the future, our scoring technique could be integrated into Eclipse's call hierarchy or a whole-program call graph visualization tool to highlight relevant suggestions. To overcome limitations in static representations, *Dora* could allow programmers to add conceptual edges [10] to the program representation, or take advantage of common programming frameworks to discern additional relationships [25].

7. CONCLUSIONS

Completing software maintenance and evolution tasks for today's large, complex software systems can be difficult, often requiring considerable time to understand the system well enough to make correct changes. To help programmers more effectively explore programs for software maintenance tasks, we present a technique that exploits both program structure and lexical information. The experimental results demonstrate that our integrated lexical- and structural-based approach is significantly more effective than a state-of-the-art structural program exploration technique, motivating further development of integrated structural and lexical techniques.

Acknowledgments

We thank Martin Robillard for his help with Suade, and David Shepherd and Jayson Hill for their help in generating the experimental data. We also thank Keith Trnka, Sara Sprenkle, Sreedevi Sampath, Thomas Fritz, and the members of HiperSpace for their helpful comments on this paper.

8. REFERENCES

[1] P. D. Allison. *Logistic Regression Using SAS: Theory and Application*. SAS Institute, Inc., Cary, NC, USA, 1999.

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

[3] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 681–682, 2006.

[4] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal of Software Maintenance and Evolution*, 16(3):151–185, 2004.

[5] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[6] B. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, Dec. 1976.

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

[8] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 112, 1999.

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

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

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[12] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[13] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[14] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 108–124, 1997.

[15] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, 2005.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[17] IBM. Eclipse IDE. online, 2007. http://www.eclipse.org.

[18] M. R. Jakobsen and K. Hornbæk. Evaluating a fisheye view of source code. In *CHI '06: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 377–386, 2006.

[19] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-oriented*

Software Development, pages 178–187, 2003.

[20] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 159–168, 2005.

[21] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 126–135, 2005.

[22] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, 2006.

[23] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.

[24] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, 2004.

[25] A. Michail. Browsing and searching source code of applications written using a gui framework. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 327–337, 2002.

[26] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[27] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[28] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 170, 2002.

[29] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[30] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS – an eclipse plug-in for source code exploration. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC '06)*, 2006.

[31] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with google. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 334–338, 2006.

[32] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 11–20, 2005.

[33] M. P. Robillard and W. Coelho. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[34] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.

[35] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An empirical study of the concept assignment problem. Technical Report SOCS-TR-2007.3, School of Computer Science, McGill University, June 2007. `http://www.cs.mcgill.ca/~martin/concerns/`.

[36] M. P. Robillard and F. Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *eclipse '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 65–69, 2005.

[37] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-oriented Software Development*, 2007.

[38] V. Sinha, D. Karger, and R. Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Visual Languages and Human-Centric Computing (VL/HCC 2006)*, 2006.

[39] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[40] M.-A. D. Storey and H. A. Muller. Manipulating and documenting software structures using shrimp views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, 1995.

[41] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[42] C. van Rijsbergen. *Information Retrieval*. Butterworths, second edition, 1979.

[43] F. W. Warr and M. P. Robillard. Suade: Topology-based searches for software investigation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 780–783, 2007.

[44] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[45] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, 2006.