Suade: Topology-Based Searches for Software Investigation

Frédéric Weigand Warr and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{fwwarr, martin}@cs.mcgill.ca

Abstract

The investigation of a software system prior to a modification task often constitutes an important fraction of the overall effort associated with the task. We present Suade, an Eclipse plug-in to automatically generate suggestions for software investigation. The goal of Suade is to increase the efficiency with which developers explore the source code by recommending locations that are likely to be relevant to the task. Based on a context of software elements (fields and methods) explicitly specified by a developer, Suade automatically generates other elements that are likely to be relevant given the context, by analyzing the topology of structural dependencies in a software system.

1. Introduction

When trying to understand a software system in the context of a modification task, standard search features, such as the Eclipse[1] Search, typically provide an initial way for developers to discover program elements related to the task. However, basic lexical search tools are often insufficient to identify all the code and artifacts that need to be investigated to complete the task. As an example, we can consider the case of jEdit 4.1,[2] an open-source text editor that supports line folding.[3] For a developer wishing to discover how the folding feature is implemented, a natural first step would be to search the code base for elements whose identifier contains the string "fold" or some variant. However, such a search returns well over one hundred types, methods, and fields, less than half of which are actually related to the folding feature. In the case of jEdit, this number of false positives is due in part to the large number of elements containing the word "folder". Another limitation of lexical searches is that they do not identify relevant elements that do not have the input keywords in their identifier.

Developers typically complement lexical searches by performing manual investigation of the code through cross-reference queries [7]. For instance, after having identified method `expandFold`, a developer might use the Eclipse Java Search to obtain all of the callers of `expandFold`. However, this single-step, manual process can be tedious and effort-intensive, especially in cases where there are few lexical cues to help the developer choose which of the search results are likely to be related to the task.

A potential strategy to help guide developers as they explore source code is to provide them with recommendations that could help steer them towards code elements relevant to a task. One solution, proposed by Robillard [6] and implemented in the Suade plug-in, is to analyze the topology of the structural dependencies of elements that have already been identified as relevant. Given a set of elements deemed relevant (the "context"), Suade will (1) search the code base for elements that have a structural relation to these elements, (2) analyze the patterns of interactions involving these elements, and (3) rank the related elements according to heuristics applied to the topology of program dependencies involving these elements. Our hypothesis is that in many cases relevant elements can be identified more efficiently through a single Suade topology-based search than through multiple, iterative cross-reference queries. Empirical evaluation has provided initial evidence in support of this hypothesis [6].

2. Related Work

Tool support for program investigation and understanding was initially developed in the form of standalone lexical search tools (e.g., grep [1]) and program databases (e.g., CIA [2], and XREFDB [4]). Basic program search and cross-referencing tools have also been provided as part of integrated development environments for many decades (e.g., in Interlisp [9], Smalltalk [3], and Eclipse [5]).

[1] www.eclipse.org
[2] http://www.jedit.org
[3] We explain the folding feature in Section 5.

Using such basic tools effectively, however, requires a certain familiarity with the source code. Furthermore, the effectiveness of the tools depends upon the investigation skills and intuition of the user.

Over the years, a number of approaches have been proposed that increase the level of automated support for program investigation. Repository mining techniques (e.g. Zimmermann et al. [12]) provide recommendations for software investigation based on the principle that code locations that were modified together in the past are probably related. Dynamic feature location approaches (e.g. Wilde and Scully [10]) can help determine which elements are part of the implementation of a certain feature by inspecting which elements are called during the execution of the program when the feature is used. Other methods analyze a textual description of a feature to find elements with related identifiers in the code base (e.g. and Zhao et al. [11]). A more complete coverage of related work is presented in a previous paper [6].

Our topology-based search technique complements existing approaches by offering developers a middle-ground between basic search tools and highly-automated techniques. It also expands the range of search techniques available to developers by analyzing a type of information latent in a software project: the topology of dependencies between program elements.

3. Specifying Context with ConcernMapper

To generate suggestions for software investigation, Suade needs a certain amount of "context" information. In our case, the context is a set of program elements (methods or fields) explicitly specified as relevant by the user. This set of elements can be discovered using any method available. Based on our experience and observations, an initial context for software investigation is often formed by selecting elements from the results of a text search. However, other avenues are also possible (e.g., using cues present in a bug report, advice from a colleague, etc.). The usage scenario for Suade assumes the presence of a "starter set", or context, but is independent from the method used to produce the set.

To allow developers to specify the context for Suade searches, we use ConcernMapper [8]. ConcernMapper is an Eclipse plug-in that allows developers to select a subset of elements from a project's code base and organize this subset into high-level abstractions called *concerns*. ConcernMapper's main goal is to provide developers with a solution for performing tasks associated with high-level concerns whose source code is scattered throughout the software system.

Essentially, ConcernMapper provides a way to map a subset of the methods and fields in a software project to individual concerns. ConcernMapper achieves this functionality by providing an Eclipse view that displays the elements that are relevant to their respective concerns (Figure 1). Developers can create concerns, which are simply labels, and add elements to these concerns by dragging the elements into a concern from any other view of the Eclipse platform. Concern models can then be saved to a file, allowing future developers to retrieve at a later stage the list of elements that are relevant to a task.

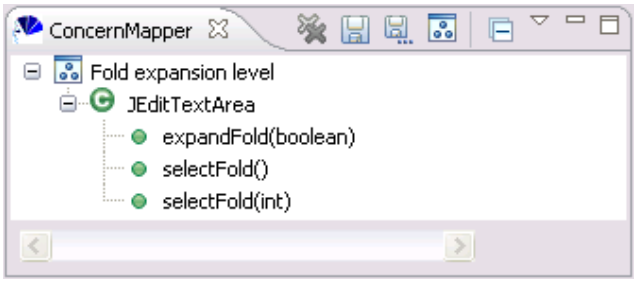

Figure 1. The ConcernMapper view showing a concern containing three methods from the `JEditTextArea` class

Each element belonging to a concern in ConcernMapper is associated with a degree value that quantitatively represents the extent to which the element is relevant to its associated concern.

4. The Suade Plug-in

The Suade[4] plug-in recommends other potentially-relevant elements in the code based on the elements tracked by ConcernMapper. The typical usage scenario for Suade is that of a software developer who does not know what elements are relevant to a software modification task. After adding a few elements that may be of interest to a new concern in ConcernMapper (or by loading a previously created concern associated with a similar task), the developer can obtain a sorted list of elements that are likely to be of interest given the elements already specified. Suade generates suggestions by building a program database, expanding the context with a list of structurally-related elements, and sorting the suggested elements according to a pair of heuristics. The process can then be iterated after the most relevant suggestions are added to the concern, providing further refinement to the set of suggestions.

4.1. Creating the program database

At the base of the Suade plug-in is a fact extraction engine that parses the source code of a project and builds a database of its elements (classes, field, methods), and their relations (e.g. field accesses, method calls). The fact extraction and program database creation functionalities of

[4]The name Suade is taken from the Latin verb *suadere* meaning "to recommend, to suggest".

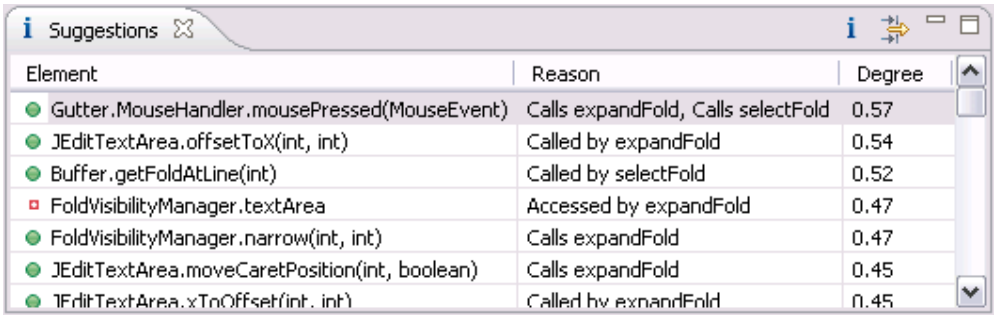

Figure 2. The Suade view containing a ranked list of suggestions

Suade are supported by the JayFX package. [5] Since JayFX keeps the program database in memory, after an initial loading phase, suggestions can be generated close to instantaneously when dealing with a code base that is not modified between iterations.

4.2. Expanding the context

When the user requests suggestions, the Suade plug-in uses the program database to retrieve all the elements that have a direct structural relation to any of the elements in the context. The relations that are used in the current implementation of Suade are *calling a method*, *being called by a method*, *accessing a field* and *being accessed by a method*. We call the elements obtained in this phase the *related elements*.

4.3. Sorting generated results

Suade ranks the results by their potential interest to the developer using a dedicated algorithm. We present only a high-level overview of this algorithm here. The interested reader can find the detailed description in a separate paper [6]. This algorithm takes into account two main characteristics of the relations between elements: *specificity* and *reinforcement*. Specificity evaluates the "uniqueness" of a relation between a context element and a related element. For example, if a context element x is related to five other elements $x_1 \ldots x_5$, and another context element y is related to two other elements y_1 and y_2, then we say that y_1 and y_2 are more specific than $x_1 \ldots x_5$. Based on the intuition that specific elements are more strongly related to the context, our heuristic ranks specific elements as more interesting.

Reinforcement evaluates the strength of the intersection between the context and a set of related element. For example, if a context element x is related to five other elements, $x_1 \ldots x_5$, and four of these elements ($x_1 \ldots x_4$) are already in the context, then we say that the remaining element (x_5 in our case), is heavily reinforced. On the other hand, if none of the elements in the set is also in the context, we do not consider the elements to be reinforced. Based on the intuition that reinforced elements are more strongly related to the context than unreinforced elements, our heuristic ranks reinforced elements as more interesting.

The algorithm starts by analyzing each relation type separately. First, we obtain, for each element in the context, the set of all elements related to it by the relation type currently analyzed. For example, for the relation type "called by", we obtain all callers of each method in the context. We then use a formula to produce, for each related element, a degree of potential interest for the element that is based on our specificity/reinforcement criterion. We then merge the results of the analysis of each relation, taking into consideration that if elements are related by several types of relations, their potential interest is greater.

In the end, our algorithm produces a single fuzzy set of elements directly related to the context whose corresponding degree (ranging from 0 to 1) represents an estimate of the element's potential relevance to the task. Furthermore, Suade also displays the reasons for which each of the elements is suggested (i.e. its relations to elements of the original context). Suade presents the suggestions are presented in a table as shown in Figure 2.

5. Example

jEdit's folding feature allows users to hide portions of text by collapsing them into single lines with a visual cue representing the fold and allowing users to expand it. By clicking on the fold marker, the user can switch between an expanded or a collapsed state. However, when in the collapsed state, clicking the fold marker will only expand one level of folding (i.e., if the expanded text has subsections that were folded, they remain folded). We posit a modification scenario in which a developer is asked to modify the folding behaviour to automatically expand every nested level of folding when a user clicks on the fold marker.

The first step for the developer is to identify a few elements that could be related to the implementation of the folding feature. In this example the developer does not have

[5] www.cs.mcgill.ca/~swevo/jayfx

access to information from previous tasks or other developers, so a new concern is created in ConcernMapper. By using Eclipse's Java search feature, the developer executes a search for elements containing "fold" in their identifiers. The search returns many results, but several are from the same class, `JEditTextArea`. The developer selects methods from this class that could be related to the task at hand based on textual cues: methods `expandFold(boolean)`, `selectFold()` and `selectFold(int)`.

All the developer has to do now to generate suggestions is to drag the concern of interest from the ConcernMapper view and drop it onto the Suade view. When the analysis completes, an ordered list of elements is presented to the developer, with the `expandFold(int, boolean)` method of the `FoldVisibilityManager` class as the 4th result in the list.

Elements suggested by Suade that are deemed of interest by the developer are then added to the initial context in the ConcernMapper view. In our example, the developer decides that the `FoldVisibilityManager`'s method `expandFold(int, boolean)` is related to the task, and adds it to the concern in ConcernMapper. Now that the set of related elements has changed, the set of elements suggested by Suade will be different. Iterating the process of generating suggestions and adding elements to the context is likely to increase the developer's chances of finding the right elements quickly. In fact, in our example the developer would be likely to have been able to complete the task by analysing the number one element suggested on the second iteration (Figure 2).

6. Summary and Future Work

To perform software modification tasks effectively, a developer must first have a good understanding of how the feature to be modified is implemented, and of where the related elements are located in the code base. Program investigation is time-consuming and error-prone. To help developers quickly find relevant elements and understand their relationships with the other elements that implement the feature of interest, we developed Suade, an Eclipse plug-in for automatic generation of suggestions for program investigation. Suade does not replace traditional program investigation techniques such as searches, call hierarchies or package browsing, but, in the absence of strong investigation cues, it does allow to focus the developer's attention to elements likely to be relevant.

We are currently working on providing incremental updates to the program database in order to reduce suggestion generation time when dealing with a modified code base. Future work will also include expanding the set of analyzed relations and providing support for an open ended set of relations.

Availability

The Suade Eclipse plug-in is free and available at http://www.cs.mcgill.ca/∼swevo/suade.

Acknowledgements

This work was supported by an NSERC Discovery Grant and an IBM Eclipse Innovation Award.

References

[1] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York, NY, USA, 1980.

[2] Y.-F. Chen, M. Y. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[3] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, MA, USA, 1984.

[4] M. Lejter, S. Meyers, and S. P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[5] Object Technology International, Inc. Eclipse platform technical overview. White Paper, 2001.

[6] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 2005.

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

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

[9] M. Sanella. *The Interlisp-D Reference Manual*. Xerox Corporation, Palo Alto, CA, USA, 1983.

[10] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7:49–62, 1995.

[11] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering*, pages 293–303, 2004.

[12] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.