Filtering out methods you wish you hadn't navigated

Annie T.T. Ying, Peri L. Tarr
IBM Watson Research Center
aying@us.ibm.com,tarr@us.ibm.com

ABSTRACT

The navigation of structural dependencies (e.g., method invocations) when a developer performs a change task is an effective strategy in program investigation. Several existing approaches have addressed the problem of finding program elements relevant to a task by using structural dependencies. These approaches provide different levels of benefits: limiting the amount of information returned, providing calling context, and providing global information. Aiming to incorporate these three benefits simultaneously, we propose an approach–called call graph filtering–to help developers narrow down the methods relevant to a change task. Our call graph filtering approach uses heuristics to highlight methods that are likely relevant to a change task on a call graph. The size of the set of relevant methods is reduced by our filtering heuristics, while global information and the calling context are provided by the call graph. We have performed two preliminary studies: a user study on identifying methods relevant to the understanding of JUnit tests on a small system, and an empirical study on how our results can help a developer perform a program navigation task with the Eclipse framework. The studies show that our approach can provide useful results: quantitatively in terms of size of the results, precision, and recall; and qualitatively in terms of finding non-trivial control-flow and being able to direct developer to the code of interest.

1. INTRODUCTION

The navigation of structural dependencies (e.g., method invocations) when a developer performs a change task has shown to be effective in program investigation [11]. Typically, only a small fraction of the structurally related elements are relevant. For example, investigating the body of program elements such as method wrappers and getters do not typically contribute much to a developer's understanding of the program.

Several existing approaches have addressed the problem of finding program elements relevant to a task by using structural dependency information. Impact analysis approaches—such as static slicing [5]—attempt to return all program elements that are relevant to a given point in the program by some criteria related to the control-flow and the data-flow of the code. Although such analyses provide information that is sound and global, the results are typically far too large for a human to understand. Call graph analyses, such as Rigi [8] and the "Call Hierarchy" view in Eclipse, attempt to return all the methods that are transitively called from a given method. The use of a graph or a tree is useful in providing the calling context for each method. However, the results are still too large even though the analyses only consider control-flow dependencies. Other approaches, such as Robillard's approach [10], use heuristics to rank the likely relevant methods based on the topology of the structural dependencies. His approach is effective in limiting the size of the results, but tends to suggest elements that are structurally close to a given method, providing a relatively local view of structurally related elements.

To augment existing approaches to help developers narrow down the program elements relevant to a task, we propose an approach that incorporates three of the goals from the existing approaches, while returning relevant results:

G1. limit the amount of information returned
G2. provide calling context
G3. provide global information

Our approach, called call graph filtering, automatically highlights the methods that are likely to be relevant to program navigation on a call graph. The size of the set of relevant methods is reduced by our filtering heuristics (*G1*), while global information is provided by the call graph (*G3*). The intuition behind the call graph filtering heuristics is that methods which do not significantly contribute to understanding the code have two characteristics in a static program call graph: (1) they are consistently closer to the leaves of a call graph for all executions (e.g., getter and setter methods), and (2) they consistently call a small number of methods for all executions (e.g., method wrappers). The results are highlighted in a call graph view we have implemented as an Eclipse plugin. Displaying the results in the context of the call graph provides the calling context of each method (*G2*).

To validate our hypothesis that the call graph filtering approach can provide results relevant to developers making a change, we have performed two preliminary studies. In the first study, we apply our call graph filtering approach to

the specific problem of identifying the set of methods that are relevant to understanding a JUnit [1] test case (MRUT). MRUTs are important to identify during a change task involving a JUnit test case because a JUnit test case may invoke numerous methods transitively, and this space of invoked methods is too large for a human to manage. Fortunately, only a small subset of these methods are likely relevant. We use call graph filtering to eliminate irrelevant methods from the set of methods that can be invoked, transitively, from a JUnit test case. We validate our approach by analyzing four JUnit test cases against the MRUTs which subjects from an empirical study have indicated to be relevant to each of the test cases. The results show that our approach can identify a small set of MRUTs, covering a good portion of what the subjects think are relevant (i.e., recall) and without a lot of noise (i.e., precision). Moreover, our qualitative analysis reveals that our approach is effective at filtering out several types of irrelevant methods to understanding a JUnit test case.

In the second study, we focus on how the results returned by our approach can be helpful to a developer performing a change task in a large system, Eclipse. We chose two real tasks we encountered during the implementation of the filtered call tree view. We found that the results returned by our approach was able to direct a developer to the relevant code when performing the tasks.

The rest of the paper is organized as follows: Section 2 describes the call graph filtering approach and its implementation. Section 3 presents two preliminary studies validating our approach. Section 4 discusses related work, followed by the conclusion in Section 5.

2. CALL GRAPH FILTERING

In this section, we walk through the design and implementation of our approach with respect to the three goals we stated in Section 1. Each of the following subsections focuses on one of the goals.

2.1 Call graph *(G3. Global information)*

Conceptually, our approach involves three steps. First, our approach takes as input a method (or a constructor) of interest. Second, our approach then produces a call graph rooted at the given element. A call graph is a graph in which a node represents a method (or a constructor) and a directed edge (`a`,`b`) represents that method `a` invokes method `b`. Finally, our approach highlights the methods that are likely to be relevant using filtering heuristics, described in the following section.

In our implementation, we use static call graphs generated by the T.J. Watson Libraries for Analysis (WALA) [2]. WALA provides static analysis capabilities for Java bytecode. The call graph analyses from WALA we use is based on the rapid type analysis (RTA)[4]. The reason behind choosing WALA and the RTA algorithm is that RTA is a practical algorithm, unlike other object or path sensitive analyses, and the WALA implementation of the algorithm reduces the deficiency of RTA by handling some common cases in an object sensitive manner, e.g., an edge from `new Thread(atm).start` to `atm.run`. We configure the call graph computation to include library calls. For example, a call to the JUnit framework `assertEquals(money1,money2)` eventually calls the application method `Money.equals`. If we had stopped expanding the call graph at `assertEquals`, which is the treatment in the Eclipse "Call Hierarchy" view, we would have missed `Money.equals`.

2.2 Filtering heuristics *(G2. Limiting result size)*

To limit the information given by the call graph, we have developed two heuristics to filter out methods in the call graph that are likely irrelevant during program investigation:

The **Don't-hit-bottom** heuristic filters out methods closer to the leaf of a call graph. Such methods include getters (a method whose sole purpose is to access a field) and setters (a method whose sole purpose is to write to a field). Inspecting the body of such methods typically do not add value to the developer's understanding of the program. We can configure the definition of "bottom" by adjusting the parameter p_{bottom}, which indicates the minimum number of methods in the callee chain for the given method to be considered as relevant.

The **Skip-small-methods** heuristic filters out methods with a small number of callees. This heuristic can filter out methods such as delegation methods which are not likely to contribute to the understanding of the application logic. We can configure the definition of "small" by adjusting the parameter p_{small}, which indicates the minimum number of direct callees for the given method to be considered as relevant.

2.3 Filtered call tree view *(G1. Context information)*

The results inferred by the heuristics are highlighted in a call tree view. The call tree view is a tree representation of the call graph. If method `a` calls method `b`, and method `c` calls `b`, then `b` would be represented as two nodes. The method's calling context, the parent of each method in the tree, is readily available in the call tree view.

We have implemented our call graph filtering approach as an Eclipse plugin. Figure 1 provides a screen shot of our tool. (The underlines, squared box, and rounded box are added to the image to assist the discussion in Section 3.2.)

3. VALIDATION

To validate our hypothesis that our call graph filtering approach can provide results relevant to developers making a code change, we have performed two preliminary studies. The first study focuses on tasks involving JUnit test case, and the second one on program navigation in the Eclipse code base.

3.1 ATM study on MRUTs

This study evaluates how good our call graph filtering applies to a specific problem: identifying methods relevant to the understanding of a test (MRUTs). We apply our approach to find MRUTs in a small application, an automated teller machine (ATM) [3]. The system contains 48 files. We validate the MRUTs of which subjects from an empirical study have indicated to be relevant to each of the test cases. The first part of the study assesses the accuracy of the MRUTs by comparing our results to the MRUTs identified by the author of the test cases. The second part of the study evaluates the interestingness of the results by study-

Table 1: Quantitative results for top 10

	precision	recall	h-mean	reduction
`transfer`	0.700	0.636	0.666	7.1
`withdrawInsufficient`	0.600	0.600	0.600	7.1
`startupShutdown`	0.100	0.167	0.125	4.9
`cashDispenser`	0.500	0.429	0.462	3.2
average	0.475	0.458	0.466	5.6

Table 2: Quantitative results for top 15

	precision	recall	h-mean	reduction
`transfer`	0.500	0.636	0.560	5.1
`withdrawInsufficient`	0.500	0.700	0.583	5.1
`startupShutdown`	0.154	0.333	0.211	3.8
`cashDispenser`	0.500	0.429	0.462	3.2
average	0.41.1	0.525	0.462	4.3

ing MRUTs that novice developers missed to identify but are correctly recommended by our tool. The rest of this section describes each of part of the study.

Part 1: Accuracy

The first part of this study involves assessing the accuracy of the MRUTs suggested by our tool with respect to the MRUTs declared by the author of the test cases. We asked the author to identify MRUTs of four JUnit test cases from the ATM system. We evaluate our results to the MRUTs identified by the author using precision and recall, two popular evaluation measures from the information retrieval community. Precision measures, of all the results returned by our tool, how much of which are the MRUTs identified by the author of the test cases. Recall measures, of all the methods the author indicated as MRUTs, how much of which are returned by our tool. To compare the precision-recall pair of measures across different result sets, we combine the two measures into one, called harmonic mean, also a popular measure from the information retrieval community. More formally, if r is the set of results returned by our tool and t is the set of MRUTs declared by the author, then precision can be expressed as $\frac{|r \cap t|}{|r|}$, recall as $\frac{|r \cap t|}{|t|}$ and harmonic mean as $\frac{2 \times precision \times recall}{precision + recall}$. In addition to the quantitative measures, we also analyzed qualitatively the types of irrelevant methods that our approach was able to filter out.

Tables 1 and 2 present the precision and recall in the two settings of the approach each of which uses a parameter setting that gives the top 10 and the top 15 results, respectively. The first column in the table lists the tests in question. Our approach achieves up to precision of 70% and recall of 63.6% for the top 10 results, and on average achieves precision of 47.5% and recall of 45.8%; the size reduction was 7.1x. As for the top 15 results, our approach achieves up to precision of 50% and recall of 63.6%, and on average achieves precision of 41.4% and recall of 52.5%; the size reduction was up to 5.1x. The precision of the test `startUpShutDown` is particularly low because many of the calls are not captured in a static call graph due to dynamic dispatch; thus, our filtering approach cannot return such calls. Using a dynamic call graph can improve the precision, and we plan to explore this as future work.

Our tool successfully filters out several types of methods that are not MRUTs:

Mock objects are used in unit tests to help isolate the part of the system to be tested, often implemented as delegation design pattern. Although a good software engineering practice, the use of mock objects can obscure the understanding of a test because such objects do not contribute to any actual functionality of the system. Our approach correctly filters out all the calls to mock objects, none of which were declared to be a MRUT by the author.

Getters and setters are methods whose sole purpose is to read from or write to a field, respectively. These methods do not contribute to the functionality of the system, but the use of these methods is a good object-oriented programming practice to encapsulate internal data in an object. Of the 37 MRUTs identified by the author of the four test cases, only one setter method, `CashDispenser.setInitialCash`, was significant to the understanding of one of the test cases. Our approach correctly eliminates all getters and setters.

Part 2: Interestingness

The second part of the study assesses the interestingness of the results returned by our approach, by analyzing what novice developers miss when they examined a test. We asked three subjects, none of whom had seen the code before, to identify the MRUTs of the four test cases. All the subjects were researchers at IBM Watson Research Center, and all of them declared that they were at least "proficient" in Java programming. The subjects were allowed to use any features from the standard installation of Eclipse for Java developers.

Our approach was able to highlight MRUTs missed by the novice developers in our empirical study. If these developers were to use our tool, they may have identified these missing MRUTs:

Retaining non-trivial control flow. Our tool can return methods that are involved in non-trivial control flow, such as forking a thread. In Java, one way to fork a thread is to call `Thread.start`. In our study, two out of three novice subjects missed to inspect the method `atm.run` and all the methods transitively called from the method. These methods the subjects neglected to examine actually form the majority of the methods invoked from a test case. When we asked the subjects why they did not inspect `atm.run` at the end of the study, they admitted that they did not know or forgot that when a thread is forked after calling `Thread.start`, the method `atm.run` is eventually invoked in the forked thread. Our approach which was able to infer `atm.run` may have helped these two subjects in reasoning about such non-trivial control flow. The "Call Hierarchy" view in Eclipse cannot return this call, although the debugger obviously can do so.

Confusion on methods with similar names. Our analysis based on structural dependencies has the advantage that the results are independent of the quality of the identifiers. Using the name of a method is a common strategy developers use to locate code of interest, but this strategy can

sometimes be misleading. In our study, one subject mistakenly reported seven calls which were not invoked at all from the test case, because the name of the test case was similar to those methods he reported. The results from our tool summarize the structural information that are transitively called by a test may have helped this subject in reasoning the methods that are possibly called from the test case.

Conclusion

Our approach was able to eliminate common types of irrelevant methods: mock objects, getters, and setters. The precision and recall of our initial prototype may seem low, but it gave a good reduction in size and it has potential to improve, for example, by using more precise call graph information from dynamic data.

3.2 Eclipse study on program navigation

The second study focuses on how the results from our approach can help a developer perform a change task in a large system, Eclipse. We chose two real tasks we encountered during the implementation of our filtered call tree view. For each task, we describe the task, how we investigated the task, and whether our call graph filtering approach can help.

Task 1

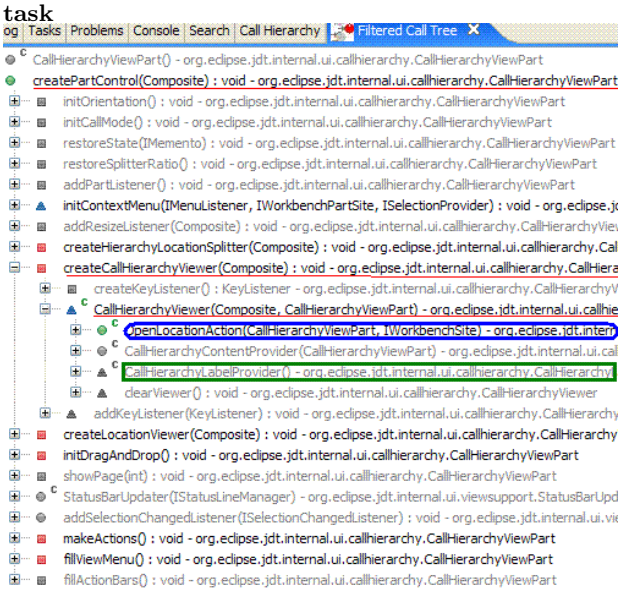
The first task involves figuring out how to display different Eclipse style images beside Java program elements depending on the modifiers on the declaration. For example, a constructor is denoted with a "C" in the image, and a public method is denoted with a green square in the image. Our initial thought was to examine the code of the Eclipse "Call Hierarchy" view, as that view has similar functionality we wanted to implement. We first guessed that the "Call Hierarchy" view would be a subclass of the class `ViewPart`, which is the abstract base class for all views in Eclipse. Indeed, we found the class `CallHierarchyViewPart` in the JDT UI project. From the class-level JavaDoc of `ViewPart`, we found out that `CallHierarchyViewPart.createPartControl` deserved further investigation as it is triggered when Eclipse creates a `ViewPart`. Thus, we could use our filtered call tree rooted at `CallHierarchyViewPart.createPartControl` to help search for the code, shown in Figure 1. We configured our approach to filter out p_{bottom}=2 and p_{small}=2 and only returned nodes in the same project (i.e., JDT UI project) and the system libraries. The method `createPartControl` [1] calls 17 methods, 7 of which highlighted by our tool. By elimination, `createCallHierarchyViewer` and `CallHierarchyView` [2] looked promising from their names. Finally, we saw `CallGraphLabelProvider` [3], the class we were looking for that encapsulates the display of labels on Java elements.

Task 2

The second task involves figuring out how to open a Java editor given a Java program element. Similar to the first task, we wanted to examine the code of the Eclipse "Call Hierarchy" view as the view has similar functionality we want to implement. Our strategy was to try to look for the registration of an UI trigger associated with the view,

[1]underlined in Figure 1
[2]both underlined in Figure 1
[3]squared boxed in Figure 1

Figure 1: Filtered call tree view on the Java label task



and in there we would be likely to find the code that opens the Java editor. Again, we started with the `CallHierarchyViewPart.createPartControl` in the JDT UI project, and we investigated the same path[4] as in Task 1 to `CallHierarchyViewer.createCallHierarchyViewer` as the creation of the viewer may contain the registration of the UI trigger. Following this path, we saw `OpenLocationAction` which worthed investigating for two reasons: the "action" part of the name could imply that `OpenLocationAction` [5] is an Eclipse action[6], which is a UI trigger; the "OpenLocation" could mean opening an editor, although we were not very certain. Investigating the body of the `OpenLocationAction` class, we found what we were looking for in this class: a call to open a Java editor.

Conclusion

From the two tasks we examined, we have shown that our call graph filtering approach was able to direct to the code we are looking for in a change task. However, there are several assumptions for our approach to work. First, we need to know which method the call graph would be rooted on. Second, when expanding the call graph, the developer must further filter out possible candidates, for example, by inspecting the name of a method.

4. RELATED WORK

Suggesting related program elements

Robillard has proposed to recommend methods of interest based on the neighbouring structurally related program elements specified as interesting by the user [10]. Their approach is very effective in limiting the amount of results

[4]The path contains methods underlined in Figure 1.
[5]round-boxed in Figure 1
[6]The Eclipse action mechanism allows actions to be added to different menus automatically.

returned and can take multiple seed points. The general hypothesis of exploiting the topology of the call graph is the same as ours. However, we use a different intuition of using the global topology of the call graph in addition to the topology of the neighbours of a given program element. To find relevant elements structurally far from the interest point using his approach, the user has to iteratively refine this interest set and reapply the analysis until one of its elements has become structurally close to an unknown target method. In addition, the results are shown in a list without calling context. It would be interesting to explore integrating Robillard's heuristics to our filtered call graph view.

Impact analyses such as slicing (e.g., [5]) try to identify all statements in a program that might affect the value of a variable at a given point in a program by analyzing the data-flow and control-flow of the source code. Slicing approaches can provide sound information about code related to a given point in the program, but they suffer from practical limitations. The results from slicing are often very large. A recently proposed approach called thin slicing addresses the large size of a slice by limiting the slice to only the statements with a value dependency the seed point [12]. Thin slicing is effective to help in tasks dependent on the data flow, e.g., locating bugs given the location of the crash; while our approach is useful in tasks that relies on control flow, e.g., navigating API of a framework data-flow of the framework intended to be encapsulated.

Test understanding

Marschall attempts to find the methods a unit test focuses on [7]. Our notion of MRUTs used in the validation is similar to that of Marschall, but with several major differences: Marschall only focuses on unit tests, whereas our approach can apply to any kinds of tests or methods in general. In addition, our call graph filtering approach can return relevant methods that are transitively called from a test, whereas Marschall only analyzed the direct calls from a test.

Xie et. al. purposed an approach that helps a user reason test cases by classifying them into two categories: tests exhibiting special cases and common cases [14]. Their results can help developers catching special cases or even common cases they had missed to test. Even after understanding whether a test exhibit special or common, a developer can use our approach to assist them understand the tests.

Change impact analysis correlating tests and code

Chianti finds affected tests given a change in the source code, by finding the changes that caused behavioural differences in the tests [9]. Chianti is subsequently used to classify whether a change caused a failure indicated by a failing test [13]. Our approach differs from theirs in purpose: their approach requires a change to trigger the tool to find the part of the change that induces the failure, our tool is targetted to assist navigation which is more exploratory in nature.

Jones et. al. proposed a technique to visualize the statements in a program according to whether it participated in failing tests only, in passing tests only, or in both passing and failing tests [6]. Our approach differs from their technique in purpose: their approach provides a summary of test results, whereas our technique provides a summary for navigation.

5. CONCLUSION

In this paper, we have presented our approach of call graph filtering to help a developer identify pertinent methods from the sea of structurally related program element. Our approach is based on simple filtering heuristics on a call graph, aiming to limit the amount of information returned to the user, provide calling context of the methods, and provide global information. We have shown some initial evidence that our approach can provide useful information: our approach achieves good precision and recall on identifying methods relevant to the understanding of tests, on the basis of what the author of the test cases declared. In addition, our approach can filter out several kinds of irrelevant methods, such as mock object calls, and retain interesting calls that are non-trivial to subjects in our study. Our validation also shows that our approach can direct developer to the code of interest in a large framework, Eclipse.

In the future, we would like to extend our work in three directions: more evaluation on both the effectiveness of the filtering heuristics and the effectiveness of the highlighted call tree view UI; exploring other filtering heuristics; and exploring different UIs.

6. ACKNOWLEDGMENTS

Thanks to Steve Fink, Tim Klinger, Paul Matchen, Jason Smith, and Rosario Uceda-Sosa for many insightful discussions; and Steve Fink for the help with WALA.

7. REFERENCES

[1] JUnit: http://www.junit.org/index.htm.
[2] WALA: http://wala.sourceforge.net/.
[3] ATM application: http://www.math-cs.gordon.edu/courses/cs211/atmexample/.
[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, 1996.
[5] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE TSE*, 17(8), 1991.
[6] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.
[7] P. Marschall. Detecting the methods under test in java. Bachelor thesis, 2005.
[8] H. A. Müller and K. Klashinsky. Rigi - a system for programming-in-the-large. In *ICSE*, 1988.
[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, 2004.
[10] M. P. Robillard. Automatic generation of suggestions for program investigation. In *FSE*, 2005.
[11] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE TSE*, 30(12), 2004.
[12] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing, In *PLDI*, 2007.
[13] M. Störzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *FSE*, 2006.
[14] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *ISSRE*, 2005.