Identifying Extract Method Opportunities Based on Variable References

Mehmet Kaya
Department of computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
mkaya@syr.edu

James W. Fawcett
Department of computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

Abstract—Long methods are usually difficult to read and comprehend due to the length and complexity of the code. As a result, maintenance can be time consuming and costly. One strategy to lower overall cost of software development for large systems is to produce smaller and less complex methods through method refactoring. This paper presents a new technique to automate the selection process of program fragments for refactoring. The soundness of this technique has been demonstrated through experiments on several different software systems. Long method defects can effectively be resolved by extracting code fragments identified with the support of a tool we have developed.

Keywords—Extract Method Refactoring; Long Method defects; Placement Tree

I. INTRODUCTION

After the delivery or release of a software product, software development enters what is known as the maintenance phase where the focus is “to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.” [9]

For large software systems, effective maintenance is difficult without readable, well-structured, and relatively simple code. If code fails to meet these properties, refactoring is one strategy to improve the internal structure of the code and thus its maintainability. Refactoring is an umbrella term to describe any process that enhances the program’s internal structure without changing its external behavior.

Extract Method refactoring resolves the Long Method defect by decomposing long methods into smaller, more meaningful or cohesive ones. For many cases, problems in large software systems can be attributed to large methods [1]. On the other hand, smaller methods are easier to read, comprehend, and maintain. The decomposition process simplifies a program by constructing more meaningful methods. Although shorter functions improve readability, the key to reducing complexity is to identify and extract, as a unit, related code fragments within a function.

Methods after refactoring are effective, since they extend the lifetime of programs by making the reader better able to understand the purpose of each method [1] and more effectively test, as well. Extract Method refactoring consists of two major activities: identification of code fragments to be extracted followed by the extraction of the identified code as a new function and replacement of the original fragment with a function call.

The extraction itself is easily achieved using built in functionality within an IDE such as Visual Studio or Eclipse. However, the selection process is less straightforward. There is not a single identification strategy and various heuristics or guidelines for selection of code fragments have been proposed in literature [4,5,6,7]. Implementation of a selection process can prove challenging and the addition of visualization to these techniques is crucial for effective execution.

Most programming languages, such as C++, C#, C and Java, obey a set of inclusion rules that govern how key constructs are placed within source code. In this paper, the constructs that we will focus on are functions and controls. We will see that placement structure of controls within a function is a tree which we will refer to as a placement tree where each element has only one parent. The approach proposed in this paper constructs a placement tree that contains variable reference counts for individual controls or scopes within a given function. The placement tree is then visualized so that each scope node has a color associated with its dominant or most referenced variable. The goal is to create functions with a single color placement tree. Therefore, a node that does not match the color of its parent node is identified as a candidate code to be extracted. In our earlier paper [3], variable references and dependencies were used for extraction of classes out of an existing class. This research showed that reference counts of variables and scope definitions alone can find effective extract method refactoring candidates.

In this paper, Section 2 covers related works, Section 3 explains the placement tree structure, which is key to this approach. In section 4, the concept of dominant variables is explained and the identification process of candidate code fragments is given in Section 5 along with an explanation of how we visualize methods to identify candidate code fragments. The focus of Section 6 is on the extraction process of identified code fragments. Experiments and results are discussed in section 7 and finally concluding remarks about our paper is presented in Section 8.
II. RELATED WORK

Program slicing has been used extensively to identify code fragments for extract method refactoring [4]. However, implementation can be tedious and time-consuming since this technique usually requires the user to manually select the slicing criterion. Furthermore, there is no a priori approach for choosing the slicing criterion for extract method refactoring. In this paper, we introduce a fully automated selection technique and placement tree visualization tool for detecting candidate extraction fragments in large scale systems.

In [4], a program slicing technique is proposed to extract code fragments related to the computation of a given variable and the state of the given object. A mechanism that uses block-based program slicing to automatically extract methods in object-oriented programs is proposed in [6]. A transformation technique to decompose functions into smaller ones, Tuck, is proposed in [7] based on program slicing.

Control Flow Graphs (CFG) are also used widely as a method extraction tool. In [8], an automatic process for extracting methods based on an input CFG of a function and a set of pre-selected nodes is introduced. To find extract method refactoring candidates, an approach based on Data and Structure Dependency (DSD) graph and longest edge removal algorithm is proposed in [5].

The technique proposed in this paper aims to identify scopes that carry out distinct operations. As shown in this paper, density of variable references in code fragments or scopes can be a good indicator for identification of main tasks in large methods. An effective refactoring method should be supported by a visualization tool or technique for the developers to observe the suggested refactoring better in large scale systems. To visualize the structure or placement tree of the input method, a treemap approach is adopted in this research.

Treemap visualization is an effective way of presenting hierarchical information where nodes are represented by nested boxes [10]. Treemap visualization in this research represents nodes on the placement tree using boxes where a parent includes all its children. The code fragments chosen for method extraction are decided based on the colors of these boxes which are appointed according to the dominant variables in scopes.

III. PLACEMENT TREE

In the proposed technique, an input method is represented with a placement tree where each node represents a different scope. Scope is defined by all code enclosed within braces, e.g., "{" and "}". A scope may include multiple other scopes and this hierarchical placement of scopes constitutes the placement tree.

In this approach, placement tree nodes are classified using eight different scope types. In Table 1, these scope types are shown with their statement coverage. The function scope itself constitutes the root node on the placement tree. If-else blocks are treated as single scopes, since the if part and else part cannot be split into two different methods.

In Figure 1, an explanatory method is shown. Figure 2 shows placement tree for the given example code where each node is identified by its start and end line number. Final visual representation of the placement tree as a treemap is also shown in Figure 2.

There are some studies that target only the second step in the subject refactoring method: extracting predefined code fragments from original method. Given an arbitrary set of pre-selected statements, in [8], an algorithm is introduced to extract them preserving the semantics of the original code. In [2], a methodology is given to extract a set of marked statements that are difficult to extract due to the presence of some certain key words.
IV. DOMINANT VARIABLES

After constructing the placement tree for the input method, this approach identifies the dominant variable in each scope. Variable with the highest reference count is identified as the dominant variable for the given scope.

Set theory will be used to better explain how we construct scopes and determine their corresponding dominant variables. \( V(F) \) is the set of all variable names that appear anywhere in the function. Long Method defects can be detected in Object Oriented Programs, Procedural Programs or Legacy Codes using the technique proposed in this paper. All global variables and data members of the class that method under analysis belongs to, may be elements of \( V(F) \). Local variables declared in the method are also in the set of \( V(F) \). In other words, all variables accessible from within the given method are possible elements of the set \( V(F) \), subject to whether they are referenced in the method at least once.

\[
V(F) = \{ v_1, v_2, ..., v_n \} \quad (1)
\]

We also have defined several functions to express various properties of program statements, variables, scopes and the method. We will use these functions in our explanations throughout the paper. Table 2 shows the functions and their respective input and output values.

**TABLE II. DEFINED PROPERTIES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>LN</td>
<td>A program statement, ( S )</td>
<td>Line # of ( S )</td>
<td>LN(( S ))</td>
</tr>
<tr>
<td>RC</td>
<td>A variable name, ( v ); and a scope, ( B )</td>
<td># of references of ( v ) in ( B )</td>
<td>RC(( v, B ))</td>
</tr>
<tr>
<td>SLN</td>
<td>A scope, ( B )</td>
<td>Starting line # of ( B )</td>
<td>SLN(( B ))</td>
</tr>
<tr>
<td>ELN</td>
<td>A scope, ( B )</td>
<td>Ending line # of ( B )</td>
<td>ELN(( B ))</td>
</tr>
</tbody>
</table>

Let \( F \) represent the set of all scopes in the given method. And every scope in the given method is represented with a set of statements, \( B \). Let the set \( V(S) \) represent the set of variables that are used for execution of statement \( S \) and let the set \( V(B) \) represent the set of all variables that appear in scope \( B \) at least once.

\[
V(S) = \{ v | v \text{ is used in } S \} \quad (2)
\]
\[
B = \{ S | SLN(B) \leq LN(S) \leq ELN(B) \} \quad (3)
\]
\[
\forall S \in B: \exists B : SLN(B) \leq SLN(B') \leq LN(S) \leq ELN(B') \quad (4)
\]
\[
V(B) = \{ v | \exists S \in B: v' \in V(S) \} \quad (5)
\]
\[
F = \{ B | \forall S \in B, B' \in F, B \neq B'; S \notin B' \} \quad (6)
\]
\[
\forall S, B: V(S) \subseteq V(F) \text{ and } V(B) \subseteq V(F) \quad (7)
\]

From formulas given above, one can conclude that every statement belongs to only one scope or node in the placement tree. A statement \( S' \) is and can only be element of one scope \( B \) that encloses statement \( S' \); but \( S' \) is not and cannot be considered as an element of an ancestor node of \( B, AB \), at the same time.

After defining our scopes and elements of them, now we find the variable that dominates the computations for every scope. We determine the dominant variable based on their respective reference counts or their respective number of appearances in the subject scope. Let \( D(B) \) represent the dominant variables in scope \( B \).

\[
D(B) \subseteq V(B) \text{ such that } \forall v \in V(B), v' \in D(B), v' \neq v:\nRC(v', B) > RC(v, B) \text{ and } RC(v', B) = RC(v'', B)
\]

The set \( D(B) \) therefore includes only those variables whose reference counts are highest in block \( B \). Every dominant variable name in the whole method is assigned a unique color to represent its power on the scope that it dominates. In our visualization tool, these colors are used to distinguish the nodes that should be extracted.

When the number of dominant variables is zero for the scope \( B \), that is \( |D(B)| = 0 \), in the placement tree, that node is represented with the color of black. On the other hand, when the number of dominant variables is greater than one for the scope \( B \), that is \( |D(B)| > 0 \), two approaches are proposed: Parent Protection, Sibling Collaboration. The idea behind these approaches is to keep the blocks that are dominated by the same variable together as much as we can.

A. Parent Protection

When scope \( B \) is dominated by more than one variable, according to Parent Protection approach, dominant variable of \( B \)'s parent node, \( BP \), is checked. If dominant variable of the parent node is an element of \( D(B) \), then parent node, \( BP \), protects its child node \( B \), and this dominant variable is assigned to be the dominant variable of node \( B \). Otherwise, one of the dominant variable from \( D(B) \) is randomly selected to be the dominant variable of this scope. Let \( dB \) and \( dBP \) represent the dominant variables of the nodes \( B \) and \( BP \) respectively.

\[
\text{if } dB \in D(B) \quad dB = dB, \quad \text{if } dB \notin D(B) \quad dB = dBP
\]

B. Sibling Collaboration

When scope \( B \) is dominated by more than one variable, according to Sibling Collaboration approach, dominant variables of its sibling nodes, \( SB \) and \( SA \), are checked. \( SB \) and \( SA \) are those nodes that come right before, and after scope \( B \) in source code (left and right nodes in placement tree respectively). Dominant variable of the sibling \( SB \) is evaluated first. If dominant variable of the sibling node, \( SB \), is an element of \( D(B) \), then sibling nodes, \( B \) and \( SB \), collaborate. In this case, dominant variable of \( SB \) is assigned to be dominant variable of node \( B \).

If dominant variable of the sibling node, \( SB \), is not an element of \( D(B) \), dominant variable(s) of the other sibling node, \( SA \), is evaluated. If \( SA \) has only one dominant variable, that is \( |D(SA)| = 1 \), and this dominant variable of \( SA \) is an element of \( D(B) \), then sibling nodes, \( BP \) and \( SA \) have collaborated, and this dominant variable is assigned to be the dominant variable of node \( B \) as well.

If \( SA \) is dominated by more than one variable, that is \( |D(SA)| > 1 \), a random variable from \( D(B) \cap D(SA) \) is chosen and assigned to be the dominant variable of both \( B \) and \( SA \). If \( |D(B) \cap D(SA)| = 0 \), a random variable from \( D(B) \) is chosen and assigned to be the dominant variable of node \( B \). Let \( dB, dSB \)
and $dSA$ represent the dominant variables of the nodes $B$, $SB$ and $SA$ respectively.

$$
\begin{align*}
\text{if } dSB \in D(B) & \text{ then } dB = dSB, \\
\text{if } dSB \notin D(B) \text{ and } |D(B) \cap D(SA)| > 0 & \text{ then } dB = dSA = v, v \in D(B) \cap D(SA) \\
\text{if } dSB \notin D(B) \text{ and } |D(B) \cap D(SA)| = 0 & \text{ then } dB = v, v \in D(B)
\end{align*}
$$

Adopting one of these two approaches, only one dominant variable is assigned to each node in our placement tree. This variable will be the one that involves in the computation of that scope most.

V. IDENTIFYING CANDIDATE CODE FRAGMENTS

There are two types of scopes that we suggest to be extracted from original code as new methods.

1. Large code fragments with a color different from parent's color. In Figure 3, we show an example for this case.

2. Consecutive sibling nodes with the same color. In Figure 4, we show an example for this case

In this paper, refactoring suggestions aim to generate methods with minimum number of color diversity. We suggest to extract first the out most scopes with a color different from their parent's color. After refactoring, resulting code should be analyzed again for further refactoring until possibly all generated methods and the original method have only one dominant variable for every scope. Therefore; the resulting code will yield methods that handle only one smaller and less complex task. Figure 3 and Figure 4 show some of the possible scopes that suit our refactoring suggestions.

VI. EXTRACTING CODE FRAGMENTS

After determining the code fragments for refactoring using our tools, developers are now to extract those fragments as new methods and replace the fragments with function calls to the new methods. Once the code fragments for refactoring are identified, extracting them is usually trivial except a few points that require attention. This section explains how the extracting process should be carried out considering some important cases that, without careful handling, might cause compilation errors or alteration in the behavior of the system.

A. Parameters of Extracted Methods

Analysis and visualization tools described in this paper have been tested on several methods with different sizes. And extract method refactoring is applied on identified fragments. Main motivation of this work is to effectively detect code fragments for extraction as new methods. Number of arguments, that needs to be passed to the extracted methods, have not been considered.

C++ is one of the most difficult programming languages for static code analysis because of its complex syntax. For this reason, for our experiments, methods that are written in C++ programming language are chosen. In C++ language, there are various ways of passing arguments to methods. By default, arguments to methods are passed by value. When an argument is passed to a method by value, changes that the method does on this argument never affect the value of the argument in the calling method. C++ also provides an option to pass arguments by reference. A reference to a variable is simply an alias for that variable. When an argument is passed by reference, the changes made on the argument within the method are also reflected to the calling method.

If a variable is never used after a code fragment, that is identified as a candidate for extract method refactoring in the original method, this variable can be passed to the method by value. Variables, that have been used after the fragment to be extracted in the original method, have to be passed to the extracted method by reference. Therefore, we suggest to use "pass by reference" whenever the programming language allows, to simplify the process of extract method refactoring.

B. Return Values From Extracted Methods

As stated earlier, IDEs like Eclipse and Visual Studio support extraction of a set of preselected statements as a new method. These IDEs have some limitations when extracting certain types of code. For example, Eclipse requires the selected code fragments not to include any return statement. Visual Studio, similarly, puts some limitation on the code with return statements. When selection contains return statements, all paths are expected to be terminated by a return statement too.

Code fragments with return statements need to be handled carefully especially when not all paths in an identified code fragment are terminated by a return statement. This is one of the greatest challenges and limitations in extraction process of method refactoring. We will expand on this problem in a future publication that we are currently working on.


C. Bound Blocks

There are two key words presence of which precludes extraction of the code fragments that they reside in. These key words are “continue” and “break” that respectively carries a loop to next iteration or halts the loop. If a code fragment contains one of these two key words and the corresponding loop is not included in the fragment, then this code fragment cannot be considered for extract method refactoring. In other words, if a code fragment or node in the placement tree contains one of the key words, “continue” and “break”, then this node is bound to its ancestor node that represents the loop associated with these keywords.

In Figure 5, we show an example code fragment that cannot be considered for extraction as a new method. The code fragment between lines 601 and 605 is bound to the code fragment between line numbers 591 and 606. Whenever two such scopes are bound, they have to be moved together in case of refactoring.

VII. EXPERIMENTS AND RESULTS

We have run our analysis and visualization tools on several methods from different systems. Identified code fragments in these methods are then extracted as new methods following the process explained in Section 5. Throughout the experiments, parent protection approach explained in Section 3 is adopted. Some of the programs and restructured versions used in experiments can be found at [12].

A. Experiments from Our Analysis Tool

We first applied this technique to a method in our tool. This method basically analyses a statement to find data declaration and references in the statement. Figure 6 shows the placement tree of the original version of the analyzed method.

We extracted three new methods and ran our tool again after refactoring. Figure 7 shows the placement trees for the methods we generated. During refactoring, code fragments with small lengths (usually two lines of codes) were not extracted, as our main focus is on extracting large code fragments.

For all the test cases that we used from our analysis tool, we could easily come up with meaningful names for the extracted methods based on their respective tasks. This demonstrates that our approach, with a high probability, will identify fragments that have a distinct task in the larger operation of the whole method. That is, suggested approach can identify fragments that actually should be separate methods.

B. Experiments from Open source Projects

Method used for this experiment is taken from a research project written by a group we collaborate. This method basically implements a part of reconstruction process of medical images obtained using cone beam and/or parallel beam collimators. The original function before any processing or refactoring has nearly 400 lines of codes with comments and white spaces. Figure 8 shows a portion of placement tree for the original version of this method.

We extracted nine methods and ran our tool again on these methods after refactoring. Figure 9 shows the placement trees for these methods. After refactoring the method has less than 40 lines. This improves the readability of the code a lot and makes the code more comprehensible reducing its complexity. Hence the overall maintainability of the whole system is improved as the developer has the chance to work on smaller and less complex methods after refactoring.

The second method used for our experiment is one of the longest methods from this research project with nearly 4000 lines of codes with comments and white spaces. Our tool clearly detects many code fragments as extract method refactoring candidates and as shown in Figure 10, one can easily observe candidate code fragments for extraction. Figure 10 shows just a portion of the placement tree for this method. Other parts of the placement tree are not any less diverse in terms of colors of nodes or their dominant variables.

Another software that we used for our experiments is called Notepad++. Notepad++ is an open source code editor and
Notepad replacement that supports several programming languages and natural languages [11]. Analyzed method, feedGUIParameters, has more than 800 lines of code. Figure 11 shows a part of the placement tree for this method. As shown in Figure 11, our tool was able to identify large code fragments that are candidates for extract method refactoring.

![Figure 8 Experiment 2](image)

![Figure 9 Refactoring result for Experiment 2](image)

VIII. CONCLUSION AND FUTURE WORK

In this paper we mainly focus on identification of code fragments for extract method refactoring. Our identification process, as stated earlier, is based on placement tree and variable reference counts in each node of this tree. This approach is straightforward to implement and it effectively works in real software systems as shown in the experiments.

In this work, initially we did not target refactoring to reduce total number of statements in systems by detecting and removing duplicated code. Yet, our visualization reveals that variable reference counts can also be used for this purpose. As shown in Figure 8, we have encountered quite a lot recurring patterns in our placement trees. Such recurring patterns can be found in Figure 9 as well in the placement trees of the new methods. When we compared the corresponding code fragments of these recurring nodes, we saw that some of these code fragments were identical to each other, while for some, there was a tremendous similarity between corresponding code fragments, although they were not identical. This shows that our approach with some improvement can be used to detect duplicated code as well. This shapes the future direction of our research together with a study to minimize the number of parameters that extracted methods require.

![Figure 10 Experiment 3](image)

![Figure 11 Experiment 4](image)

REFERENCES


