Identifying Extract-method Refactoring Candidates Automatically

Tushar Sharma

Siemens Corporate Research and Technologies, Bangalore, India.
tushar.sharma@siemens.com

Abstract

Refactoring becomes an essential activity in software development process especially for large and long life projects. *Extract-method* is one of the most frequently used refactorings to address code smells such as long and incohesive methods, and duplicated code. Automated means of identifying opportunities for refactoring can make the software development process faster and more efficient. In this paper, an abstraction for methods viz. *Data and Structure Dependency (DSD) graph* and an algorithm viz. *longest edge removal algorithm* to find extract-method refactoring candidates is proposed.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering

General Terms Algorithm, Design

Keywords Refactoring, Automatic refactoring candidate identification, Extract-method refactoring, Data and Structure Dependency graph, Longest edge removal algorithm

1. Introduction

In the current era of software engineering, refactoring established as a common practice among software developers. Opdyke [7] coined the term *refactoring* and defined it as "behavior preserving transformations". The technique is practiced to improve the maintainability of a software system. Periodic refactoring reduces technical debt towards software design and hence prepares the software design for smooth future extensions and bug fixes.

The process of refactoring essentially includes identification of a candidate spot and a specific refactoring to apply, followed by source code modification to carry out the identified refactoring. The first step i.e. identification of a candidate spot is a challenging task, which is in general performed manually by analyzing source code/design. The refactoring process can be made easier, faster, and cost effective by employing automated means to identify candidate spots for specific refactorings.

Extract-method refactoring is one of the most frequently used refactorings to improve the software quality. In an empirical evaluation, Dirk et al. [10] identified extract-method as the most frequently occurring refactoring. Similarly, Mika et al. [5] discovered that the driver to carry out extract-method refactoring is the most mentioned driver in their analysis to discover refactoring drivers. Hence, it is a necessary requirement to identify candidate spots for extract-method refactoring automatically.

Rest of the paper is organized as follows: Section 2 discusses related work, Section 3 introduces Data and Structure Dependency graph and Section 4 explains the overall methodology. Later, in Section 5 I discuss the need of structural dependencies in this context and then I apply the proposed method on an example in Section 6, and finally I conclude in Section 7.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WRT '12 June 01 2012, Rapperswil, Switzerland.

Copyright © 2012 ACM 978-1-4503-1500-5. . . $10.00

2. Related Work

A large body of work in literature for method extraction is based on program slicing. Lakhotia et al. [4] proposed a program transformation method called *Tuck* based on program slicing. Similarly, Maruyama [6] proposed block-partitioning algorithm to decompose control flow graphs. Moreover, Komondoor et al. [3] proposed a methodology to extract marked statements using control flow graphs. In addition, Tsantalis et al. [9] identified extract-method refactoring candidates using block based slicing.

The first drawback of slicing based solutions in the context of refactoring candidate identification is that the user has to provide slice-criterion (statement(s) or variable) for each method, hence these solutions are not fully automated. Secondly, these algorithms produce results based on the selected criterion. It might be error-prone for large software systems since it depends on the experience of the developer and complexity of the software system. Further, these algorithms are not exploiting the fact that a logical block of statements are inter-related with each other. Hence, slicing based on a variable may not result in a proper extraction. Furthermore, one physical statement block (used in block based slicing) might not entirely cohesive together i.e. a physical block may have multiple logical cohesive blocks. Due to this, block based slicing algorithms may not result in optimal extract-method candidate identifications. Therefore, better algorithms are required to achieve effective extract-method refactorings.

3. Data and Structure Dependency (DSD) Graph

DSD graph is a combination of a data flow graph and a structure dependency graph. A DSD graph shows data dependencies as well as structural dependencies among source-code statements. A DSD graph can be defined as follows:

A DSD graph G can be defined as a pair (V, E), where V is (V_d, V_s) and E is (E_d, E_s), where V_d is a set of data-vertices (vertices representing source code statements), and V_s is a set of structure-vertices (vertices representing a block of statements), where E_d is a set of data-dependency edges (edges between the vertices V_d; $E_d \in \{(u, v) | u, v \in V_d\}$), and where E_s is a set of structure-dependency edges (edges between a data vertex and a structure

vertex; $E_s \in \{(u, v) | u \in V_d, v \in V_s$ or $u \in V_s, v \in V_d\}$). An edge e_d (where $e_d \in E_d$) exists between two vertices u, v when there exists a data dependency between vertex u and v. An edge e_s (where $e_s \in E_s$) exists between two vertices u, v when there exists a structural dependency (one statement is dependent on the structure defined by the other statement) between vertex u and v.

DSD graph shows data dependencies among source statements and structural dependencies between block statement (such as 'if', and 'for') and statements contained within the block.

DSD graph provides an abstract view of a method. It shows information relevant to identify refactoring candidates (data and structural dependencies) while leaving out other details. The combination of data as well as structural dependencies prepares a firm ground on which effective refactoring candidate identification algorithm can be formulated and executed.

The next sub-section discusses about the difference between DSD graph and data/control flow graphs. Later, Section 5 explains the rationale behind having structural dependencies in DSD graph.

3.1 Difference between DSD graph and data/control flow graph

Techniques based on *data flow graph* [8] and *control flow graph* [2] has been used extensively in static analysis and its relevant applications. These techniques are extended further in this paper to purpose a novel solution to the stated problem. The proposed DSD graph is different from data/control flow graph as argued below:

- **Encapsulation of data as well as structural aspects**: DSD graph encapsulates data dependency as well as structure dependency of a method. At the same time, data/control flow graph captures only one aspect (data or control) in a graph. This encapsulation enables us to apply an effective algorithm to identify extract-method candidates.
- **Abstraction**: In general, data/control flow graphs can be employed at different levels of abstractions. At the lowest level, control flow graphs define a block as a linear sequence of statements with one entry and one exit points and define flow of control among these blocks. On the other hand, DSD graph is designed to address a specific set of problems which require statement level information. Hence, a data node in DSD graph strictly represents a source-code statement.

4. Identify Extract-method Refactoring Candidates Automatically

The proposed work follows three steps mentioned below to identify extract-method refactoring candidates automatically.

4.1 Label the method

In order to create DSD graph of a method under analysis, it must be labeled first. Integers starting from 1 are used to label a method. Following guidelines are used to label a method:

- The labeling starts from parameters of the method. A number is assigned to each parameter of the method.
- Each local variable declaration is assigned a label.
- A label is assigned to statements consisting local variables and parameters. However, statements with only class member variables and control statements (such as continue, break, etc.) are not allotted any label. They are treated as part of the last statement.
- For each block of statements ('then' and 'else' part of an 'if' statement, 'for' loop, and while loop, etc.); a label is allotted. Each block may have associated expression (for example, terminating condition in do-while loop), which is combined with the block; hence block and the associated expression gets a single label. The number allotted to a block is always greater than the number allotted to any statement in that block. This label is used to show structural dependencies while creating DSD graph.

4.2 Generate DSD graph

Following guidelines need to be pursued to create a DSD graph for a method:

- Each labeled statement is considered as a node of a DSD graph.
- Determine a data dependency set for each node. Draw an edge between two nodes to represent data dependency. A node (B) is dependent on another node (A) if node A declares or modifies a local variable which is used in node B.
- Create a structure-node for each block statements ('then' and 'else' part of an 'if' statement, 'for' loop, and 'while' loop, etc.). A structural dependency edge connects a structure-node to all other nodes contained within the block.

4.3 Identify sub-graphs to apply extract-method refactoring

Once a DSD graph is created, a strategy needs to be applied to identify sets of tightly-coupled statements which can be extracted out as new methods. The work proposes an algorithm viz. *longest edge removal algorithm* to identify such sets.

A long method could be performing too many tasks. Each task is implemented by a set of statements logically connected. Data dependency within these logical blocks tends to be high. Two statements connected by a data-dependency edge and placed closer in a DSD graph is likely to be more coupled than two statements connected by a data-dependency edge and placed farther in the DSD graph. Hence, longer the edge, lower the likelihood that the connected nodes reside in a single method. Thus, eliminating longest dependency edge results in a set of disconnected sub-graphs, which are reported as candidates for extract-method refactoring.

The proposed algorithm is given below:

```
//Input: DSD graph of a method
LongestEdgeRemoval(DSDG aDSDG) {
  //look for any disconnected sub-graph
  sub-graphs = findDisconnectedSubGraphs(aDSDG)
  for(sub-graph, for all sub-graphs)
    identifyCandidate(sub-graph)
  }
//Input: DSD sub-graph of a method
//Output: Boolean value indicating extract-method
//candidate
boolean identifyCandidate(DSDG aDSDG) {
  //number of statements in the graph should not
  //below a threshold
  if(aDSDG.countStatements()<MIN_STMT_THRESHOLD)
    return false
  //number of parameters in the sub graph should
  //not exceed a threshold
  if(aDSDG.countParameters() >PARAMETER_THRESHOLD)
    return false
  //Find out the longest edge of aDSDG
  longestEdge =findLongestEdge(aDSDG)
  //if the longest edge (between 'source' and 'sink'
  //nodes) is only edge, then do not remove it.
  if (singularEdge(longestEdge)) {
    longestEdge.processed = true
    return identifyCandidate(aDSDG)
    }
  //notionally remove this longest edge from aDSDG
```

```
longestEdge.remove()
longestEdge.getSinkNode().addParameter
  (longestEdge.getDependencyVar())
//look for any disconnected sub-graph emerged
//due to removal of an edge
sub-graphs = findDisconnectedSubGraphs(aDSDG)
candidateIdentified = false
for(sub-graph, for all sub-graphs) {
  b=identifyCandidate(sub-graph)
  if(b) {
    extractMethod(sub-graph)
    sub-graph.processed(true)
    candidateIdentified = true
    } }
if(candidateIdentified)
  return false
else
  return true
}
```

Listing 1: Longest edge removal algorithm

The algorithm accepts a DSD graph for a method and looks for existing disconnected sub-graphs. If the algorithm finds one or more disconnected graph initially, disconnected sub-graphs other than the main-body sub-graph are ideal candidates for extract-method refactoring (since there is no dependency among these sub-graphs). If there is no additional disconnected sub-graph present (other than main-body sub-graph), then the algorithm notionally removes a longest edge within the sub-graph and checks the emergence of any disconnected sub-graph due to the removal of the edge. In this algorithm, weight of the edge represents the length of the edge, where edge-weight = (label of the sink node - label of the source node). Here, the *notional removal of an edge* is defined as follows: we tag the edge as removed and put the variable in parameter list to satisfy the data dependency. If the algorithm finds a disconnected sub-graph which fulfills parameter-count and statement-count thresholds, then the algorithm is applied on the sub-graph recursively to explore the possibility of smaller sub-graphs. If smaller sub-graphs cannot be found (based on statement-count threshold) then the disconnected sub-graph is reported as a candidate of extract-method refactoring.

4.3.1 Behavior preservation

Behavior preservation is an important property of refactoring. The algorithm takes care of the property by introducing appropriate parameters while removing longest edge. The algorithm computes two types of parameters viz. 'in' and 'out'. The algorithm determines 'in' parameters for the emergent sub-graph while notionally removing a longest edge. All unique dependencies are treated as 'in' parameters for a sub-graph. All local variables which are getting modified in a sub-graph are treated as 'out' parameters. If a parameter is common in sets of 'in' and 'out' parameters, then that is treated as an 'out' parameter only. These parameters help a developer achieving behavior preservation while applying the longest edge removal algorithm.

Similarly, the algorithm defines a few constraints to achieve behavior preservation. For example, if a condition block is split into multiple disconnected sub-graphs then either all sub-graphs must be within the conditional statement, or all sub-graphs must be enclosed with the conditional statement.

5. Need of Structural Dependence

As stated earlier, DSD graph encapsulates data as well as structural dependencies. It is natural to capture data dependencies in the context of extract-method refactoring. However, this section describes the need to capture the structural dependencies in this context.

Let us consider an example and apply the proposed algorithm on the data dependency graph of the example. Listing-2 shows the considered example along with labeled statements.

```
void func(int a)
  {           1
  2.  int i=getFirstPrime();
  3. while(a>i) {
  4. int b=a;
  5.  int c=process(b);
  6. int d=process(c);
  7. print(d);
  8. if(!b%i) {
  9. i=getNextPrime(i);
  10. b=process(b);
  11. print(b);
      }}
  12. print(i);
  }
```

Listing 2: Labeled source code of the example method

Figure 1(a) shows data dependency graph for the considered example.

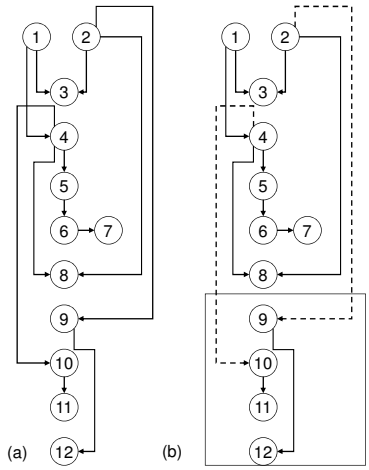

Figure 1. (a) Data dependency graph for considered example. (b) Longest edge algorithm applied on the data dependency graph.

Now, on the application of the longest edge removal algorithm, we get a disconnected sub-graph (node 9 to 12) as perceived in Figure 1(b). In this figure, dotted lines represent notionally removed data dependency edges. As described earlier, all disconnected sub-graphs can be extracted as new methods. However, extracting this sub-graph to a new method would be erroneous because the new method would violate behavior preservation property. Hence, it is necessary to incorporate structural dependencies along with data dependencies to achieve extract-method refactoring automatically without violating behavior preservation property.

6. Example

A method is taken as an example from an open source project CppCheck [1] and applied the proposed methodology on it. Labeled version of the method is given below in Listing 3.

```
static void array_index(const Token *tok, std::
```

```
                                        1
list<ExecutionPath *> &checks, unsigned int
                    2
varid1, unsigned int varid2) {
   3                    4
5. if(checks.empty()||varid1 == 0|| varid2 == 0)
     return;
   // Locate array info corresponding to varid1
6. CheckBufferOverrun::ArrayInfo ai;
7. ExecutionPathBufferOverrun *c=dynamic_cast<Exe
     cutionPathBufferOverrun *>(checks.front());
8. std::map<unsigned int, CheckBufferOverrun::
     ArrayInfo>::const_iterator it;
9. it = c->arrayInfo.find(varid1);
10.if (it == c->arrayInfo.end())
     return;
11.ai = it->second;
   // Check if varid2 variable has a value that
   //is out-of-bounds
12.std::list<ExecutionPath *>::const_iterator it;
13.for(it=checks.begin();it!=checks.end(); ++it)
     {
14.ExecutionPathBufferOverrun *c = dynamic_cast
     <ExecutionPathBufferOverrun *>(*it);
15.if (c && c->varId == varid2 && c->value >=
     ai.num[0])
       {
       //variable value is out of bounds,report
16.    CheckBufferOverrun *checkBufferOverrun =
         dynamic_cast<CheckBufferOverrun*>
           (c->owner);
17.    if (checkBufferOverrun)
         {
18.      std::vector<unsigned int> index;
19.      index.push_back(c->value);
20.      checkBufferOverrun->arrayIndexOutOfBounds
           (tok, ai, index);
         break;
         }}}}
```

Listing 3: Labeled source code of the considered example

The DSD graph for the example method is constructed and the longest edge removal algorithm is applied on it. After some iterations, the algorithm found a disconnected sub-graph (node 6 to 23). The algorithm further analyzed the identified sub-graph to identify smaller disconnected sub-graphs for extract-method candidates. Consequently, the algorithm reported two candidates (node 6 to 11, and node 12 to 22) for extract-method refactoring as shown in Figure 2. In this figure, black lines represent data-dependency edges, red lines represent structure-dependency edges, and dotted black/red edges represent notionally removed edges.

As described earlier, the algorithm computes 'in' and 'out' parameters for the refactoring candidates. For the first sub-graph, the algorithm identified two 'in' parameters viz. *checks* and *varid1* and one 'out' parameter viz. *ai*. For the second sub-graph, the algorithm identified three 'in' parameters viz. *ai*, *varid2*, and *tok*.

7. Conclusions and Future Work

The paper proposes a new method to identify extract-method refactoring candidates automatically. It proposes a new abstraction *Data and Structure Dependency (DSD) graph* to model a method and an algorithm viz. *longest edge removal algorithm* to identify extract-method refactoring candidates automatically. The proposed method is applied and one example is presented in this paper to show the effectiveness of the proposed method.

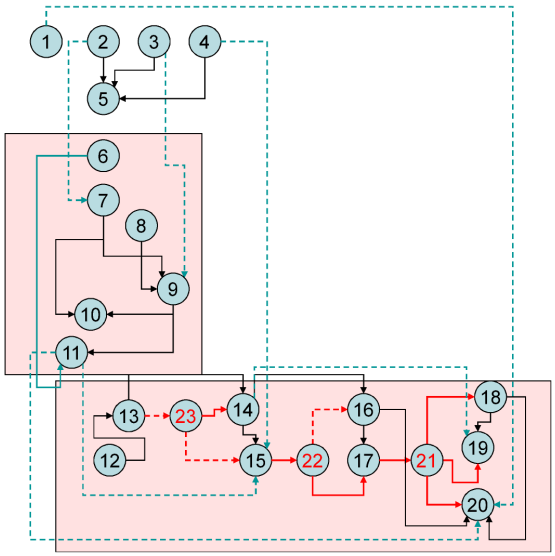

Figure 2. Two sub-graphs identified as candidates of extract-method refactoring on application of longest edge removal algorithm.

In future, I would like to develop a tool to realize the proposed method and carry out comprehensive case-studies to validate the approach.

References

[1] Cppcheck. URL `http://cppcheck.sourceforge.net/`.

[2] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479.

[3] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 33–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1883-4.

[4] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. In M. Harman and K. Gallagher, editors, *Special Issue on Program Slicing*, volume 40 of *Information and Software Technology*, pages 677–689. 1998.

[5] M. V. Mäntylä and C. Lassenius. Drivers for software refactoring decisions. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 297–306, New York, NY, USA, 2006. ACM. ISBN 1-59593-218-6. doi: 10.1145/1159733.1159778.

[6] K. Maruyama. Automated method-extraction refactoring by using block-based slicing. *SIGSOFT Softw. Eng. Notes*, 26:31–40, May 2001. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/379377.375233.

[7] W. F. Opdyke. Refactoring: A program restructuring aid in designing object-oriented application frameworks. Technical report, Ph.D. thesis,1992.

[8] W. Stevens, G. Myers, and L. Constantine. Classics in software engineering. chapter Structured design, pages 205–232. Yourdon Press, Upper Saddle River, NJ, USA, 1979. ISBN 0-917072-14-6.

[9] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 119–128, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3589-0. doi: 10.1109/CSMR.2009.23.

[10] D. Wilking, U. Kahn, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica Software Engineering Journal*, 1(1), 2007.