Source Code Comprehension Analysis in Software Maintenance

Nedhal A. Al-Saiyd
Computer Science Department, Faculty of Information Technology
Applied Science Private University, Amman-Jordan
e-mail: nedhal_alsaiyd@asu.edu.jo

***Abstract*—Source code comprehension is considered as an essential part of the software maintenance process. It is considered as one of the most critical and time-consuming task during software maintenance process. The difficulties of source code comprehension is analyzed. A static Bottom-up code comprehension model is used. The code is partitioned into functional-based blocks and their data and control dependencies that preserve the functionality of the program are analyzed. The data-flow and control-flow graphs reflects the dependencies and assist in refactoring process. The proposed strategy helps in improving the readability of the program code, increase maintainer productivity, and reducing the time and effort of code comprehension. It helps maintainers to locate the required lines of code that constitute the functional area that the maintainers are searching for in their maintenance work.**

Keywords-source code comprehension; code comprehension models; refactoring; effort estimation; source line of code (SLOC)

I. INTRODUCTION

Software maintenance (SM) is a critical activity of software evolution, when maintainers continue to correct, enhance, add or update existing software functionalities to adapt to business and environment changes. Software functionality depends on the number of functions; which provides services and reflects software architecture in accordance to the requirement specifications. SM is classified into four main categories: corrective, adaptive, perfective, and preventive maintenance. It becomes increasingly difficult and costly task as software systems continue to grow. Understanding software system is a prerequisite task before doing any maintenance process [1].

The source code lines of programs can be used in many years in use-maintenance cycles, when it is adapted to work in the changing environment. Maintaining programs that are written by others is difficult task when reading or analyzing them. Much of the time is spent in trying to understand carefully the source code, its structure characteristics and functional dependencies, which in turn increase the estimated time and effort required for maintenance. Large size of software source code is positively correlated to high cost estimation. The comprehending source code is considered to be a time consuming task, where software maintainers spend 50% or more of maintenance time on understanding the source code [2], [3], [4], [5]. Modern software systems become of large size, have complex structure and require great effort from maintainers to understand them in a timely manner. Maintainers try to use code comprehension strategy to limit the amount of source code that is needed to read and modify. However, they spend relatively long time in searching the source code to locate the intended part of code that are relevant to the maintenance task at hand [6].

Software maintainers classified code comprehension into two classes:

- Functional approach: is interested in what the source code does.
- Control-flow approach: is interested in how the source code works.

Source code comprehension defines the links between application domain and the source code, and specify the code structure, and behavior [7], [8], Source code comprehension makes software maintenance easier and more effective, and improves the reusability of existing software. It tries to extract and represent the knowledge about several aspects of the program depending on the task of interest [2]. This research is focused on the problem of source code comprehension and the identification of the relevant source code parts that are most relevant to the intended changes when adding a new feature or modifying existing features of large-scale software systems. This needs to analyze the code statements, identify the logic interrelationship of code statements and extract the semantic information from the low-level abstractions. The syntactic knowledge help the maintainer to form a semantic structure of a problem into multi-leveled knowledge. This syntax and semantics models are generated.

This paper is organized as follows. Section II present the main code comprehension models. Section III overviews related work. Section IV defines the difficulties of source code comprehension. The proposed methodology is introduced in section V. Section VI provides conclusions.

II. CODE COMPREHENSION MODELS

The process of source code comprehension can be classified into four code cognitive models [9]:

1. *Top-down code comprehension model*; the knowledge of the program domain is restructured, and mapped to the source code in a top-down manner. The process starts with a general hypothesis about the nature of the program, which represent a high-level abstraction or concepts of the program. Then, the general hypothesis is examined, refined and verified to form subsidiary hypotheses in a hierarchical layout. Each hypothesis represent a segment or chunk of program code. The low-

978-1-5386-0539-4/17/$31.00 ©2017 IEEE

levels are generated continuously until comprehension model is achieved. This model is used when the maintainers are familiar with the code.

2. *Bottom-up code comprehension model*; the maintainers read the complete source lines of code, and then group these lines into chunks of higher-level abstractions. The elementary program chunks are specified based on the control flow model, and the procedural relations among chunks are defined based on functional relations. The knowledge of high-level abstractions are incrementally grouped until a highest-level of program understanding is achieved. The comprehension is enhanced using refactoring of code functionalities. Bottom-up comprehension chunk the micro-structure of the program into macro-structure and by cross-referencing these structures. Bottom-up comprehension is less risky than a bottom-up strategy, where the lower-level hypotheses can be identified directly from concrete source code.
3. *Hybrid or Knowledge-based code comprehension model*; the comprehension knowledge about the code is grasped from the integration of bottom-up and top-down models, because the maintainer navigates through the source line of code and jumps through different chunks when searching the code to find the links to the intended block of code. The understanding of the program evolves using the maintainer's expertise and background knowledge together with source lines of code and documentation.
4. *Systematic and as-needed strategies*; on which the maintainers focus only on the code that is related to particular evolution task. The maintainers use a systematic method to extract the static knowledge about the structure of the program, and the causal knowledge about interfaces between different parts of the program at execution time. In the systematic macro-strategy, the programmer traces the flow of the whole program. This strategy is less feasible for large programs, more mistakes could occur because the maintainer miss some important interactions.

Nowadays many open-source and closed-source software utilize wide range of requirement specifications, functionalities, architecture designs, algorithms, idioms and source code fragments (code clones) that are written in different programming languages and operate on different platforms. The software developers and maintainers can use them in their source line of code. Due to the utilization of well-analyzed, designed, and tested artifacts; the quality of the maintained software will be improved, while maintenance time, effort and cost are reduced significantly; comparing to writing the programs from scratch [10].

III. Literature Review

Many researches work on analyzing the structure of source code alone or work on analyzing the concepts of the comments and identifiers within code and found their influence on the code understandability. In early researches (in mid-1980s to mid-1990) of program comprehension, the source code is analyzed and the data-flow and/or control flow graphs extracted, where operations are represented by nodes in the directed and cyclic graph and the edges represented the control flows and data flows between operations. The produced graphs represent the code-level information. They considered code comprehension task as NP-hard characteristics. Therefore, to resolve this problem heuristic-based approaches appeared that used concept-recognition approach. It uses the low-level of abstraction such as program statements, condition, variables declarations, loops, modules, etc., were combined within a knowledge base to form higher level events [11], [12]. During the parsing of the source, the series of lower-level programming events are organized in tree structure. The low-level events are integrated to form the higher-level of abstraction that represents the data structures. Completing all program events represent the program understanding. This approach is applied using limited size of Pascal programs

In [13], [14], the high-level of abstraction are derived from the low-level of program events using set of rules during program parsing and the syntax trees are generated. The higher-level of abstraction concepts and language concepts are generated from the low-level events and are combined to control flow and data flow information. This approach applied on large size COBOL programs.

Woods and Yang [11] worked on dividing the programs into series of programming blocks and represented them as nodes into block graph. The edge in the graph represents the data flow between the two nodes. The structural and knowledge constraints between program blocks are derived from the data-flow and control-flow information that are produced in the pre-processing phase; as intermediate results, together with the syntax graph of the program. In this approach, various program understanding complexity were analyzed.

In later researches, information retrieval techniques or informal token-based approaches are used to define function names, variable names and comments. The clusters of elements that share common sets of attributes are identified. It is better to integrate the understanding of the program source code with the understanding of informal program token-based information, because it combines the different levels of abstraction and perspectives to increase the level of understanding and to form a complete view of program knowledge. It integrates the code-level information together with domain-level information.

IV. Difficulties of Source Code Comprehension

A deep knowledge of the source code is required for better understanding the problem domain and solution models. There is a wide-range of factors that makes comprehension difficult and raises maintenance effort estimation. The source code programs are studied extensively from different perspectives. They are concerned with program code to improve the code comprehension. We extract and aggregate many factors practically from different programs written using different programming languages and have different sizes:

- The source code is dynamically changed over time and the characteristics of the program are changed. Also, the

program quality is declined and the programming language can be outdated when it became older.

- After many times of maintenance, software may contain many source code fragments that are logically similar to each other with varying degrees, which are commonly called "*code clones*" [15], This increases dependency among code segments and may cause defective code.
- High-complexity of source code structure characteristics (cyclomatic complexity) measured by high coupling and low cohesion.
- The application domain is complex. The maintainer faces difficulties in transferring the conceptual models into specific formal models.
- Programming language domain knowledge has difficulties; relating to programming language expressions and syntactic rules.
- The existence of large number of input and output identifiers that are propagated throughout the source code. Therefore, it becomes harder to maintain the code when there is large number of interactions between of large-scale software elements.
- Lack of up-to-date documentation availability; analysis models, design models.
- Absence of meaningful comments in the source code, which is considered as semi-structured textual data.
- The large-size of the module measured by number of line of code or number of function points within each module.
- Maintainer has low experiences, skills, low familiarity with the program code, platform and the programming language. Maintainer spends more time and effort trying to search for the intended part of software application in maintenance tasks.

Maintained software application is generally composed of adding new lines of code, changing lines of code, enhancing the software quality, or reusing lines of code from other sources in the software under maintenance. The successful maintenance needs to comprehend and adjust code. Code comprehension has its influence on the time and effort, and it is difficult to predict accurate comprehension time and effort.

The software maintenance process is influenced by the program comprehension process. More than 47% of maintenance efforts are spent in code comprehension process. If we have m software modules and we need to modify '*k*' modules of them To modify '*m*' modules then there is a need to check '*N*' interfaces among the changed and the affected modules [16], N is expressed as:

$$N = k \times (m - k) + k \times \left(\frac{k-1}{2}\right) \quad (1)$$

where:

K: the number of modules modified

N: the number of module interface checks

Hence, to modify 20% of the modules, it needs to re-test 38% of the modules interfaces.

V. The Proposed Methodology

The proposed method focuses on extracting the important properties of source code to achieve a better understanding of the software.

The source code of different programming languages are studied and analyzed from different viewpoints theoretically and experimentally to extract data different levels of abstraction. The maintenance can occur in different parts and level of source code, and it is affected by implicit dependencies among functions, classes, inherited classes, and access techniques. Therefore, the implicit dependencies are needed to explore and understand the relationships among code segments.

A. Collecting Significant Data

Program comprehension is considered as a knowledge-intensive activity, where the maintainers use and emphasis on sufficiently great amount of knowledge. To explore and understand the software system, the following data are derived based on the static information (i.e. collect code data without execution).

- The invocation point to a particular function.
- The arguments and results of a function.
- The control flow sequence of operations to reach a
- particular location in code.
  Identifying a particular set of variable, definition locations and usages.
- Naming and declarations of abstract data types.
- Discover clones; the multiple implementation of the functionality in different code fragments in the same program. Code clones perform similar function. To locate them, functional analysis is required.
- Identify the functional access points for a particular data object. Identifying the inputs and outputs of each unit code.
- Identify the relationships among different parts and components of the software, and capture coupling and cohesion of classes. Identify what are the effects of any modification that may cause.
- Cluster and classify the similar source code components to form the blocks of code.
- What are the documentation? extract more information about Analysis models and design models
- Identify the comments locations and explain their concepts.

B. Code Comprehension Activities

At any time during a program comprehension process, the maintainer can apply any code comprehension model, when the size and the complexity of the program structure are varied from one program to another and from block to another. To achieve comprehension the following activities need to be followed, where figure 1 illustrate the general activities:

a. Read the Code and Documentation (if any): Reading the source code line-by-line or in any arbitrary order to understand the workflow and the application behaviour of the program, and this assist to locate the code where

the change should be performed. Also, read and eexamine the application documentation, which is preferred to be consistent and up-to-date; requirements and design models and specifications. But unfortunately, the documentation of many systems are not available or are outdated.

b. **Execute White-box and/or Black-box** of the program to inspect the input, sequence of implemented functions, output and the consequences. It is depends on the application type and the maintainer skills and knowledge. Also, the graphical user interface plays an important role in exploring the intended parts of code. In some cases, it may be considered as starting points in order to explore the application behaviour and be familiar with the functionality. The dynamic runtime information is acquired from the executed program.
c. **Extract block:** the source code is partitioned conceptually into coherent block of code (i.e. segments of code) that share common functionality. Then analyze what each block works (functional approach) and how each block works (control approach). Partitioning into blocks depends on the continuous statements that shared common functionality and code attributes. The block may have more than one related functions. The structure of functions within the block of code is defined. This will facilitate the allocation of particular functions and their statements within a block. Partitioning helps in improving the readability of source code, filtering irrelevant code, locating data structures, assisting maintainers when locating the intended code in one area, and saving the maintenance effort and time.
d. **Write a meaningful comments;** when the maintainer wants to modify the code and the source code program contains relatively few comments, or failed to match the concepts of the related code. The comment is written in the source code associated with the block-level and function within a block. Structuring the code provides deep knowledge in several levels of abstractions.
e. **Analyze The Internal Structure Dependencies**: The inheritance hierarchies and functional dependencies between areas are analyzed to preserve the intended behavior of the source code. Analyzing data and control dependencies help in identifying the code clones and remove them.
f. **Generate Program Graphs;** by transforming the source code of the program into functional graphs. The data-flow graph and control-flow graph (that is similar to that produced in white-box testing) assist to identify the data and control dependencies in the source code. They make it easier for maintainer to read, understand and find which parts is needed to maintain and which parts are affected by the maintenance.
g. **Refactoring:** this favorable technique is used in code comprehension through analyzing the implicit structural dependencies and find out the data and control interrelationships. UML design class diagrams are used to assists in identifying the locations of declaration and usages code elements (i.e. attributes names, types, and visibilities, the methods calls, the passing messages, and the class inheritances). UML design class diagram and design sequence diagrams show the dynamic and sequential methods dependencies in class-class and/or superclass-subclasses. Refactoring technique improves interactively the software structure, rename the methods and attributes, eliminate the conflicts, and reserve the functionality of the program. Refactoring process also eliminate the code clones.

The software architecture is standardized to have low-coupling and high-comprehension to decrease complexity of source code. Using a meaningful and readable naming of functions, methods and data helps maintainers to facilitate the understanding ability of the code and to locate the intended parts that need maintenance.

h. **Generate different documentations formats** are also generated that can help the maintainer in better understand the code and better organize and perform maintenance tasks, especially when the documentation are not complete or not available.
i. **Search for the intended parts to maintain,** where control-flow data flow graphs that are produced from functional refactoring assist to allocate them.

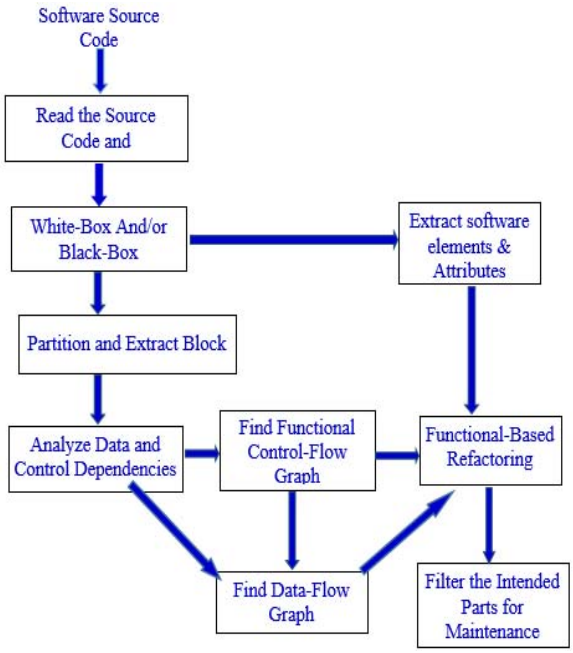

Figure 1. Source Code Comprehension Framework.

VI. Discussion and Conclusions

Obviously, code comprehension is a time-consuming task. More time is needed to produce deeper knowledge and precise results by applying static code analyzer; to facilitate the understanding of code. Code comprehension efforts

strongly depends on the maintainer skills and experiences, the programming language, the program size (either measured in SLOC or function points), the adopted maintenance model, the declaration of data types, range of data values, program structure, hierarchy of code parts dependency, and the static function-call graph at statement, function and class level. There accurate measurements for the code comprehension since the same maintainer given different program each time, but the 5 different case studies with similar code complexity and the same programming are given to 10 maintainers. The code comprehension estimation times are ranged approximately between 52% and 70%

To reduce the amount of time and improve the estimation level of code comprehension effort, more accurate (SLOC) predictions are required in highly-structured source code, since McCabe's cyclomatic complexity number showed a linear relationship with SLOC and one can predict the other [17]. The mechanisms to accurately measure code comprehension time and effort estimation need some skills and quantitative data to accomplish and still remain inexact. It is a challenging factor [18].

For this research, a *static Bottom-up code comprehension model* is used. The main conclusion points are:

- The proposed methodology enhances the overall readability of source code.
- The program source code is functionally partitioned into blocks at the functional higher-level of abstraction, and their interactions and dependencies are analyzed. The block have more than one related functions.
- The data-flow and control-flow graph assist to identify the data and control dependencies in the source code. Identifying data and control dependencies are done before refactoring the code attributes, which in turn enhance the design and quality of source code.
- Re-write the comments for the source code at the function-level.
- The software architecture is standardized to include low-coupling and high-cohesion to decrease complexity of source code.
- It will help maintainers to locate the required lines of code that constitute the functional area that the maintainers are searching for in their maintenance work.
- The proposed methodology contributes in better understanding the factors that correlate to code comprehension and maintenance time and effort and estimation
- It helps to better managing and supporting efficient maintenance releases.

ACKNOWLEDGMENT

The authors are grateful to the Applied Science Private University in Amman, Jordan for the partial financial support granted to cover the publication fee of this research article.

REFERENCES

[1] P. I. Tripathy. and K.Naik, Software Evolution and Maintenance: A Practitioner's Approach, Wiley, 2015.

[2] J. Belmonte, P. Dugerdil, and A. Agrawal, "A Three-Layer Model of Source Code Comprehension", Proceedings of the 7th India Software Engineering Conference ISEC 14, 2014. DOI>10.1145/2590748.2590758

[3] T. Corbi, "Program understanding: Challenge for the 1990s", IBM Systems Journal, Vol 28, No. 2, pp. 294 – 306, 1989.

[4] G. Parikh and N. Zvegintov, Tutorial on Software Maintenance. IEEE Computer Society Press, 1996.

[5] E. Nurvitadhi, W. W. Leung and C. Cook, "Do Class Comments Aid Java Program Understanding? Frontiers in Education", Volume 1, page T3C-13 - T3C-17, 2003.

[6] J. I. Maletic and A. Marcus (2001). "Supporting program comprehension using semantic and structural information", 23rd International Conference on Software Engineering (ICSE). Toronto, Ontario, Canada, IEEE Computer Society: 103-112.

[7] M. M Lehman, J. F. Ramil, et al., "Metrics and laws of software evolution-the nineties view", 4th International Software Metrics Symposium, 1997.

[8] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution", Encyclopedia of Software Engineering, P. Laplante, Ed. Taylor & Francis LLC, 2010.

[9] M. A. Storey "Theories, Methods and Tools in Program Comprehension: *Past, Present and Future",* Software Quality Journal 14, pp. 187–208, DOI 10.1007/s11219-006-9216-4.

[10] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution. Encyclopaedia of Software Engineering, P. Laplante, Ed. Taylor & Francis LLC, 2010.

[11] S. Woods and Q. Yang, "The Program Understanding Problem: Analysis and a Heuristic Approach", Proceedings of the 18th International Conference on Software Engineering (ICSE'96), IEEE Computer Society Press, Berlin, Germany, March 25-29,1996, pp. 6-15.

[12] C. Tjortjis, N. Gold, P. Layzell, and K. Bennett, "From System Comprehension to Program Comprehension", Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2000), IEEE Computer Society Press, Oxford, England, August 26-29, 2002, pp. 427-432.

[13] W. Kozaczynski and J. Q. Ning, "Automated Program Understanding by Concept Recognition", Automated Software Engineering (Special Issue on Knowledge-Based Software Engineering), Springer, Vol.1, Np. 1, Mar. 1994, pp. 61-78.

[14] W. Kozaczynski, J. Ning, and T. Sarver, "Program Concept Recognition," Proceedings of the 7th Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, McLean, VA, September 20-23, 1992, pp. 216-225

[15] E. Duala-Ekoko and M. P. Robillard, "Clone Region Descriptors: Representing and Tracking Duplication in Source Code", ACM Transactions on Software Engineering and Methodology, Vol. 20, No. 1, Article 3, June 2010, pp. 1-32.

[16] COCOMO II Model Definition Manual, Version 2.1, 1995 – 2000 Center for Software Engineering, USC. Available at: http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf.

[17] J. Seel, A Tool-Chain Approach to Source Lines of Code Estimation Using Petri Nets and Directed Graphs, A PhD Dissertation submitted to The Faculty of The School of Engineering and Applied Science of The George Washington University, May 19, 2013.

[18] Z. Zia, A. Rashid, and K. S. I. uz Zaman, "Software Cost Estimation for Component Based Fourth-Generation-Language Software Applications," Software, IET, vol. 5, no. 1, 2011.