Program comprehension from the perspective of testing

[1]*Matej MADEJA (1st year),*
Supervisor: [2]*Jaroslav PORUBÄN*

[1,2]Dept. of Computers and Informatics, FEI TU of Košice, Slovak Republic

[1]matej.madeja@tuke.sk, [2]jaroslav.poruban@tuke.sk

***Abstract*—Program comprehension is an important part of software development process. The paper looks for the ways how to simplify program comprehension and finds how the tests usage can facilitate it. Hence, we briefly define the program comprehension, test types and their relationship. Moreover, we analyze researches of similar issue and compare their approaches. Finally, future research directions are suggested.**

***Keywords*—program comprehension, testing, execution, source code, projections**

I. Introduction

Software development and in particular its maintenance is demanding from time viewpoint. Many various research projects deal with support of software development and try to accelerate this process through software development enviroments, programming tool support, languages, etc. One of the areas that arises is the comprehension of a program. Creation and modification of software can be included under the wider concept called software maintenance. During maintenance many changes are made to the code, and when working in a team, the programmer often needs to understand and modify the source code of another one. Customer requirements often mirror the problem domain that each programmer should understand to meet them in a specific implementation. However, during the implementation of a system abstraction (semantic) gap between problem and solution domain occurs [1]. The goal is to maximize speed of software development and management, therefore the reduction of the abstraction gap is crucial.

II. Program comprehension

A person comprehends a program if one understands its structure, behavior and connection to the application domain [2]. In a simplified interpretation, we can say that the program comprehension is a level as a programmer understands the existing program functionality, in order to improve functional or non-functional qualities. As many researches point out, this process tends to take up to a half of programmer's time during their work with the source code [3][4][5].

Program comprehension is an activity that helps expert programmers build a mental model of the program that includes both program-specific and domain knowledge [6]. The mental model is created in the programmer's head from defined specification. Based on the created model, the programmer creates a specific software artifact, the most common source code. The code is then compiled into an executable application, and the process is iteratively repeated. The order of actions in this process is not fixed and may change (for example, when the programmer has to understand a part of an existing program, and the new implementation must adapt the existing one). Domain understanding is the key to a successful system development [7], but for maintenance, the programmer does not need to know the entire code and the compete application context.

Another area related to software maintenance is reverse engineering. Frequently, these two areas are confused, but the difference is that reverse engineering helps to understand the program [8]. Reverse Engineering focuses on techniques and tools to create high-level abstractions of the code, or in other words it analyzes the source code to deduce design features. Program comprehension looks into the effectiveness of these tools and additional knowledge of software development practices.

III. Approaches of program comprehension

Comprehension congitive model is a description of a psychological process included in program understanding to achieve the mental model of the program [9] and we can divide it into two basic approaches: top-down and bottom-up.

Some of existing models use various combinations of these two approaches. Bottom-up comprehension models are based on the fact that during the source code reading abstract concepts are created by collecting low level information [10][11]. Understanding is achieved from the bottom up, ie. by reading the code and mentally grouping the individual rows into higher-level abstractions. This approach focuses on situations where the programmer does not understand the domain, yet. Example of this model is Pennington model [12].

On the other hand, as a reverse approach to program comprehension, several top-down models have been proposed when the programmer already had a previous (at least partial) experience with the domain. An example is the Brooks model [13]. The top-down models use the programmer's knowledge of the domain to create a set of expectations that are mapped to the source code.

However, the real use of models indicates that programmers do not exclusively rely on one approach but use their combinations [14]. They use one approach as a pre-dominant strategy based on their knowledge in the domain and slightly switches between comprehension processes. Letovsky [15] claims that programmers achieve ease of understanding by changing their strategies of comprehension with regard to external stimuli.

IV. Testing as an additional part of source code

The task of testing as part of the development cycle is to ensure the source code stability. Testing identifies defects or bugs of applications that need to be fixed. We can say that this is a process of accessing the functionality and correctness of a software through analysis [16]. Tests expose errors in the code, which can also help in understanding the source code itself. Imagine a situation where a programmer works in an unknown code and has the task to implement the necessary functionality. Interference with the source code can affect the original code undesirably – break it. Tests can alert the programmer to the appearance of error and reduce his comprehension gap. Through the tests, the programmer can understand the program better.

The main test tasks are quality assurance, reliability estimation, validation and verification [16]. The goal of each development process should be to deliver a quality product with the smallest cost (expense). Understanding the program, possibilities of the integrated development environment (IDE), programmer's experience, programming language, and all other aspects of the development process have a direct or indirect impact on the resulting product. To understand the software, we try to use everything that is relevant to the code, so we consider tests will be a great choice.

Basically, we divide the tests into *Static* and *Dynamic*. Static testing involves all types of reviews, inspections, and walkthroughs. Dynamic testing or actual validation involves all functional and non-functional testing types. Lewis [17] defines and analyzes 60 existing test techniques in his book. The most famous of them are black box, white box, grey box [18][16], regression [19], reliability, usability, performance, unit etc. The various methods that Lewis mentioned are often combined, and by combining simpler methods brings extended testing methods with new insights on software. In general, we know the following main types of testing [20]:

- Functional Testing,
- Performance Testing,
- Security Testing.

A. *Functional Testing*

The goal of Functional Testing is to verify required functionality and behavior. Test cases are created based on the customer's specifications, ie. they directly verify required functionality. Since functionalities are closely related to program comprehension (due to a narrow connection to functionality and source code), the main types of functional testing are also briefly described:

1) *Unit Testing* is the lowest level of testing mainly performed by developer to test the unit of code. The purpose is to isolate the smallest test pieces of the code and check they are working properly in isolation [21].
2) *Integration Testing* is testing where tests communicate with multiple different modules of the application. The goal is to check whether the data pass through various components correctly. The main representatives are top-down and bottom-up tests [22] (a similar approach as in the program comprehension, see Section II).
3) *System Testing* is a general test of the system in its behavior and functionality based on the requirements document. An example is regression, smoke or sanity testing.
4) *Acceptance Testing* is mainly known as alpha and beta testing. They make sure that the customer is able to perform the required functionality.
5) *White-box and Black-box Testing* and their combination (*Grey-box Testing*) [23]. Mostly used are for example Data Flow, Coverage, Basis Path and Loop Testing [24]. In back-box testing, also known as Behavioral Testing, tester does not know the internal structure, design, implementation, application, respectively functionality that is being tested. The advantage of black-box testing is that it does not require the source code directly, eliminating the need for instrumentation and source code availability [25]. On the other hand, white-box testing is the detailed investigation of the internal logic and structure of the code where the tester understands the internal structure of the system. The tester selects inputs to achieve a specific workflow and compares results with the expected output. Gray-box testing is a combination of previous ones and involves having access to internal data structures and algorithms for the purpose of designing the test cases, but testing at the user or black-box level [26].

To understand the program, we will be interested in all types of functional tests because they are linked to the source code and we will be able to use them to increase the program comprehension. It might seem that from last group of tests only white-box testing is suitable for us to understand the program, as white-box testing expects to know the internal structure of the code. It is a great assumption that the opposite approach when the programmer tries to understand the functionality of the black-box test, will increase program comprehension, too. On the other hand, white-box tests are too complex and detailed [16][26] so black-box tests that are not so closely related to the source code may be useful for easier understanding of the program, because they look at the functionality from the outside.

B. *Performance Testing*

Performance Testing is one of a non-functional testing types and includes all time related parameters like load time, access time, run time, execution time, etc. Most popular methods are Stress Testing and Load Testing. These tests have more or less a benchmark character, so they will not be interesting for us.

C. *Security Testing*

This testing type is important as to protect the information, services, skills and resources of adversaries and the cost of potential assurance remedies. Examples of these tests are Fuzz Testing, Brute Force, SQL Injection or Penetration Testing. For program comprehension they are not as important as Functional Testing, but the partial test results we could use the partial test results to improve the development process and code quality.

V. State of the art

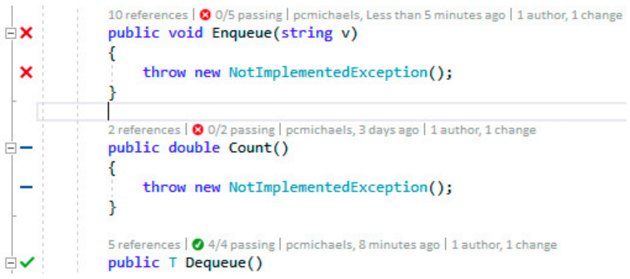
The relationship between understanding the program and testing shows a lot of research. Benedusi et al. in their paper [27] investigated the role of testing and dynamic analysis in the process of program comprehension. In their *Docket* project they explored the potential of these activities for software comprehension objectives. The paper says programmers are using product manuals and reverse engineering to understand the program's maintenance. According to the authors test cases are the passive or active subject of existing product activities (specification, test case design, debugging & error analysis, change request formulation, change analysis, regression testing) producing various kinds of peculiar and valuable knowledge. They argue that test cases are important starting points for the capture and reuse of knowledge acquired empirically and lessons learned during the operational lifetime of the product. They used manual alpha and beta tests, where these tests may not only reveal common issues, but also empirically highlight aspects of software that are the most challenging and difficult to comprehend. On the basis of this tests the programmer knows what is important to maintain the system and finds out that for system maintenance is program understanding necessary. They recommend using regression testing to increase program comprehension.

Sneed [28] considers the tester as part of the development team and in his paper he figured out that the tester is the one who needs the broadest knowledge about the system. The author describes in detail the requirements of the tester and comprehension has been used here to define a knowledge acquisition process. Testers need to identify the test objects, to determine the test cases and to assess the adequacy of their test efforts by means of a test coverage measurement. When the tester knows the system at all levels (from a global perspective) the author again brings us to the idea from the opposite perspective that in tests are very valuable information for understanding the program.

From the viewpoint of program comprehension, there is an approach of concern-oriented code projections [29] and for the programmer is provided only the selected and merged source code related to a particular concern. LaToza et al. [30] compared the code perception of experts with novices. They have found that novices have analyzed code statement by statement and wanted to understand the whole system, even if it was not necessary. On the contrary, experts used "caching" so they did not study every line of code, but they used higher abstraction. Their results lead us to source code projections which try to filter out unnecessary source code and show the programmer only the parts that are relevant.

Moonen et al. in the book *Software evolution* [31] write about the testing and evolution connection and its effect on the program comprehension. They look at software development and testing in the agile development process (Extreme programming, XP). They introduce "test-driven refactoring", or refactoring of production code, too, which are induced by the structuring of the tests. On the basis of real experience, they confirm that the extensive test suite can stimulate the program comprehension process, especially in the light of continuously evolving software, and derive the following benefits for comprehension:

- testing policy encourages programmers to explain their code using test cases,
- the requirement of 100% test success ensures the documentation is kept up-to-date,
- tests provides a repeatable program comprehension strategy,
- a comprehensive set of tests reduces the comprehension gap when modifying source code,
- systematic unit testing helps build team confidence.



```
10 references | 0/5 passing | pcmichaels, Less than 5 minutes ago | 1 author, 1 change
public void Enqueue(string v)
{
    throw new NotImplementedException();
}

2 references | 0/2 passing | pcmichaels, 3 days ago | 1 author, 1 change
public double Count()
{
    throw new NotImplementedException();
}

5 references | 4/4 passing | pcmichaels, 8 minutes ago | 1 author, 1 change
public T Dequeue()
```

Fig. 1. Unit tests results and test coverage in IDE Visual Studio 2017.

A. *The opposite viewpoint*

As we can see, above researches look at software development as a chain of understanding functionality, implementation and programming of tests. If a programmer explores the foreign code, mostly he is not interested about tests, despite all the above-mentioned researches show a close relationship with the comprehension. The problem is not with programmers, but with the tools that are used for development, mostly IDE. From our point of view, we want to focus on linking existing tests with main application source code within the IDE to bring advantage for simpler understanding the existing code.

As mentioned in the Section IV everything that helps to understand the program has the main goal of achieving high quality product. The IDE can help with understanding the program, using class browser, tests explorer, reference counts, search, icons in editor, refactoring, etc. There are small utilities that help the programmer with comprehension of source code by interconnecting with tests. An example is *Visual Studio*, where test results and their coverage are displayed directly in the source code (Fig. 1). Above the tested method is visible how many tests uses the method and how many tests were successful. The programmer can navigate and find particular test where he can analyze the issue. Before each line in the editor is displayed the icon that represents the coverage of the given line (red cross = covered by at least 1 failing test; green check mark = covered by only passed tests; blue dash = not covered). These tools are gently linked to tests, but the source code of tests has much larger potential.

VI. Future directions

All of mentioned tools and approaches for program comprehension are beneficial, our idea is in mining the context of test and linking them to the main program code. Since any research with similar idea has not been explored yet, for first we will need to perform experimental observation of tools designed to simplify program understanding (such as source code projections). Experiments will focus on the functionality and capabilities of these tools. Only then we can design new tool in conjunction with tests.

A. *Test Projections*

Dynamic concern-based projections [29] seem to be an interesting solution how to filter the source code from multiple files into one view, simple to understand for programmer. Similarly we should be able to create test projections. It means that from multiple test classes we will be able to show just needed tests. The programmer would then see different use-cases of the application.

B. The Most Common Workflows

Tests often involve calling methods in a specific order. E.g. the user sign in to the administration, goes to the user subsection, selects particular user account and changes password. This process is subject to a certain call stack of methods to be performed. We can assume the test scenarios created in the tests will be often used the same way in the production (by user). Therefore, on the test scenarios, the programmer could see most used stack of methods call and they can help to understand the context of code.

C. Web and Mobile Applications

Research is commonly realized on small or medium open-source or self-developed application that often use the basic system API, a few libraries, etc. In terms of applications in production that are much more complex, the results of such research are only partially usable, so we want to focus on larger projects. Within our university we have courses such as *Web Technologies* and *Application Development for Smart Devices*, where students develop complex applications with a lot of libraries where designed tools should be used for experiments. Accordingly, there are many open-source web applications, which means that we can explore the understanding of the already existing applications with a variety of participants (expert, novice, student) who have not seen the source code before.

Acknowledgment

This work was supported by project KEGA 047TUKE-4/2016 Integrating software processes into the teaching of programming.

References

[1] J. Greenfield and K. Short, "Software factories: Assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 16–27. [Online]. Available: http://doi.acm.org/10.1145/949344.949348

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the 15th International Conference on Software Engineering*, ser. ICSE '93. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 482–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=257572.257679

[3] T. Kosar, M. Mernik, and J. C. Carver, "The impact of tools supported in integrated-development environments on program comprehension," in *Proceedings of the ITI 2011, 33rd International Conference on Information Technology Interfaces*, June 2011, pp. 603–608.

[4] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2622669

[5] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *2014 14th International Conference on Quality Software*, Oct 2014, pp. 276–285.

[6] R. W. Collins, A. R. Hevner, G. H. Walton, and R. C. Linger, "The impacts of function extraction technology on program comprehension: A controlled experiment," *Inf. Softw. Technol.*, vol. 50, no. 11, pp. 1165–1179, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.04.001

[7] R. Offen, "Domain understanding is the key to successful system development," *Requirements Engineering*, vol. 7, no. 3, pp. 172–175, 2002.

[8] H. Müller and H. Kienle, "A small primer on software reverse engineering," 01 2009.

[9] M.-A. D. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, pp. 187–208, 2006.

[10] F. Détienne, *Software Design - Cognitive Aspects*. Springer London, 2002.

[11] N. Pennington, "Empirical studies of programmers: Second workshop," G. M. Olson, S. Sheppard, and E. Soloway, Eds. Norwood, NJ, USA: Ablex Publishing Corp., 1987, ch. Comprehension Strategies in Programming, pp. 100–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=54968.54975

[12] ——, *Stimulus structures and mental representations in expert comprehension of computer programs*. Cognitive Psychology, 1987, vol. 19, no. 3, pp. 295 – 341.

[13] R. Brooks, "Using a behavioral theory of program comprehension in software engineering," in *Proceedings of the 3rd International Conference on Software Engineering*, ser. ICSE '78. Piscataway, NJ, USA: IEEE Press, 1978, pp. 196–201. [Online]. Available: http://dl.acm.org/citation.cfm?id=800099.803210

[14] M. P. O'Brien, "Software comprehension – a review & research direction," University of Limerick, Tech. Rep., November 2003.

[15] S. Letovsky, "Cognitive processes in program comprehension," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 58–79. [Online]. Available: http://dl.acm.org/citation.cfm?id=21842.28886

[16] M. Ehmer and F. Khan, "A comparative study of white box, black box and grey box testing techniques," vol. 3, 06 2012.

[17] W. E. Lewis, *Software Testing and Continuous Quality Improvement, Third Edition*, 2nd ed. Boston, MA, USA: Auerbach Publications, 2008.

[18] S. Kumar Swain, D. Mohapatra, and R. Mall, "Test case generation based on use case and sequence diagram," vol. 3, 01 2010.

[19] S. Thakare, S. Chavan, and P. Chawan, "Software testing strategies and techniques," vol. 2, 05 2012.

[20] I. Hooda and R. S. Chhillar, "Software test process, testing types and techniques," *International Journal of Computer Applications*, vol. 111, no. 13, pp. 10–14, February 2015, full text available.

[21] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 201–211.

[22] S. Weißleder and H. Lackner, "Top-down and bottom-up approach for model-based testing of product lines," *arXiv preprint arXiv:1303.1011*, 2013.

[23] M. Ehmer Khan, "Different forms of software testing techniques for finding errors," vol. 7, 05 2010.

[24] N. Gupta, "Different approaches to white box testing to find bug," *International Journal of Advanced Research in Computer Science & Technology (IJARCST 2014)*, vol. 2, no. 3, pp. 46 – 49, July - Sept. 2014.

[25] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 523–534. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884791

[26] S. Acharya and V. P. Pandya, "Bridge between black box and white box – gray box testing technique," *International Journal of Electronics and Computer Science Engineering*, pp. 175 – 185, 2012.

[27] P. Benedusi, V. Benvenuto, and L. Tomacelli, "The role of testing and dynamic analysis in program comprehension supports," in *[1993] IEEE Second Workshop on Program Comprehension*, Jul 1993, pp. 149–158.

[28] H. M. Sneed, "Program comprehension for the purpose of testing," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, June 2004, pp. 162–171.

[29] J. Porubän and M. Nosál', "Leveraging program comprehension with concern-oriented source code projections," in *OASIcs-OpenAccess Series in Informatics*, vol. 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[30] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287675

[31] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, *On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 173–202. [Online]. Available: https://doi.org/10.1007/978-3-540-76440-3_8