Proceedings of the Fourth International Conference on Computing Methodologies and Communication (ICCMC 2020)
IEEE Xplore Part Number:CFP20K25-ART; ISBN:978-1-7281-4889-2

Moving Towards Program Comprehension in Software Development: A Case Study

Abdelaziz Fellah and Ajay Bandi
School of Computer Science and Information Systems
Northwest Missouri State University
Maryville, MO USA
{afellah, ajay}@nwmissouri.edu

Mahmoud Yousef
Dept. of Mathematics and Computer Science
University of Central Missouri
Warrensburg, MO USA
yousef@ucmo.edu

***Abstract*—The goal of this paper aims at promoting program comprehension in computer science, in particular software engineering and object-oriented programming courses. The research methodology is evaluated through an empirical study which involves an active participation audience of undergraduate and graduate computer science students. With no previous knowledge on program comprehension, participants were explicitly instructed to use their computing skills and own strategies to comprehend a set of Java programs of different difficulty levels. We did not impose on the participants any specific paradigm to comprehend the source code. We conducted three case studies with two groups of students, and the results were promising, despite of the fact that students had no previous knowledge of program comprehension. The results of this study shows that participants identified four dimensions to comprehend the source code - namely input/output activities, visual scanning, computer science knowledge, and application domain. In addition, the results provided strong evidence for the validation of the hypotheses that we formulated earlier before conducting the case studies. Another factor worth mentioning is the indentation of the code which might play a role in debugging.**

***Index Terms*—Program comprehension software engineering, program understanding, source code comprehension mental models**

I. Introduction

In a world where technologies change rapidly, software developers spend a substantial amount of time trying to understand a piece of software and comprehend its source code to meet new challenging requirements. We refer to such a discipline in software engineering as *program comprehension*. In the literature, other research terms such as program understanding and source code comprehension, are also used to describe program comprehension. For years, researchers have attempted to understand how developers comprehend the source code and numerous approaches have been proposed, such as mental and cognitive models, software visualization, empirical evaluation, top-down, and bottom-up program comprehension [2], [3], [9], [12], [17]. Most of these approaches and models are grounded in case-based studies and geared towards experienced developers. As software is often not properly documented, automated tools such as static code analysis and trace visualization are used to facilitate the program comprehension process.

Although program comprehension has been considered as an area of research, little exploration has been done to evaluate its viability as a learning programming activity in the classroom. In its most basic form, program comprehension can be introduced in a classroom setting under various formats and levels for improving student learning. Thus, the development of techniques and tools that support the program comprehension process can make a significant contribution to students' computing and programming skills. For example in programming courses, students should be able to discern an instance of an existing source code using program comprehension activities, generating basic ideas, and conveying valuable insights about such a code.

In this paper, we are not claiming that we developed a general and conclusive program comprehension framework to be adopted in the classroom. However, our goal is to add values of program comprehension to computer science courses, in particular software engineering and programming courses. This will provide students with a number of starting points, from which they gain deep insights into other avenues of software development such as maintenance and evolution. Thus, our intention is to integrate a small scale of program comprehension activities in such courses, as a software process, in order for students to navigate through the source code, and build a mental bridge between the source code and the program behavior. For example, undergraduate students in subsequent programming courses are often given programming assignments to be modified using a new language and system platform. Students spend time comprehending the programming assignment, syntactically and semantically, before they attempt to modify the program, whereas software engineering students face often the same challenge, except with larger and complex programs.

In this empirical study, the participants were undergraduate and graduate computer science students enrolled in software engineering and object-oriented programming, respectively. Student characteristics and materials that have supported this study will be discussed in the next sections. In this paper, the terms, participants and students, define the same entity and are interchangeable. Our research methodology is evaluated

through three case studies described later in the remaining of this paper. The case studies consist of three Java programs of different levels of difficulty and complemented with a series of object-oriented features. In this study, our focus was not based on any technical framework. However, our goal is to identify program comprehension patterns used by the participants. Furthermore, each Java program has its own characteristic in terms of *i.e.,* size of the code, domain, code style and complexity, attributes, number of methods, and code snippets. Consequently, each case study may involve different program comprehension activities in terms of understanding the program's inner behavior. The focus of students' attention is to comprehend each Java program and build a higher-level mental model of the source code under study. We did not impose on the participants any specific paradigm of comprehending the case studies, and students were not prescribed to any specific approach. However, they resort to their own strategies to navigate through the Java source code and decipher some idea of what the program is doing.

Based on our own experience and prior knowledge of the participants in other computer science courses, have led us to intentionally formulate a set of hypotheses in advance and then validate them at the end of this study. In addition, in order to reduce bias by a single researcher and validate the soundness of our approach, this study has been conducted by two researchers and two teaching assistants using the same settings and metrics (*i.e.* projects, participants, resources, procedure, classroom environment). The results from each individual researcher were merged and analyzed in this paper.

The structure of the paper is organized as follows. In section II, we survey the related work and research challenges that appear in program comprehension. Section III discusses the research methodology in terms of research hypotheses, participants' characteristics, case studies, survey, and data collection. Section IV evaluates and analyzes our approach, and in Section V we conclude with a summary of our methodology, and findings, and expectations in the near future.

II. Background and Related Work

Numerous program comprehension strategies have been suggested in literature, which focus on helping a developer to gain an understanding of a source code and comprehension of its program behavior. Research has shown that source code and program behavior are the most addressed parts of program comprehension [8]. In this section, we briefly summarize a number of program comprehension strategies from which a developer can choose from, without strictly predicting any particular model in advance. For example, a mental model of von Mayrhauser and Vans [13] suggests that developers build a mental or conceptual representation of the source code under consideration including the system‘s control and data flow. On the other hand, cognitive model [2], [6], [19] consists of the knowledge base of the programmer such as domain-related knowledge, language-related programming knowledge, and temporary information structures. All of these form the mental representation, which the developer builds from the code source while comprehending the system. Developers also use the assimilation process to reconstruct general knowledge links between a problem domain and a source code. For example, top-down comprehension has been used as a strategy by developers for reconstructing knowledge about the domain of the program and then map this knowledge into the source code. On the other hand, after a source code is read, bottom-up comprehension models abstract the lines of code into higher-level abstractions. For example, a limitation of top-down comprehension is that novices who are lacking domain knowledge cannot use this approach. Developers are known to mainly focus on the source code itself, rather than documentation and other artifacts [10]. Several techniques have been used to delve into the source code, such as object-oriented paradigm, feature-orientation, annotations, functional decomposition, and supporting tools. Despite such studies, it is still not definitive whether such techniques have a positive impact on the developer's capabilities to understand a program [10]. Nonetheless, our empirical study is partially inspired by these guidelines and strategies. Software visualization techniques and tools have also been proposed to assist developers for exploring the comprehension process and visualizing the run-time behavior of programs. An area of research that has not received much attention is program comprehension in the context of real-time systems. The software space of program comprehension has been extended to real-time systems in [4], [5]. Furthermore, two orthogonal and hybrid paradigms, timed event component comprehension and timed event program comprehension, have been introduced and complemented with an empirical IoT irrigation case study [4]. Software visualization models, in particular graph-visualization tools (*i.e.,* Imagix 4D, Klocwork, SHriMP) have also been developed to augment existing evaluation mechanisms in the context of program comprehension [1], [11], [14], [15], [18]. Overall, visualization ambition is apparent in program comprehension, but has limited success because of the level of abstraction details and depth to be viewed. These views are based on the characteristics of the domain model that incorporates both behavior and data [7], [16].

III. Methodology

A. Research Hypotheses

Our initial research hypotheses (*i.e.,* predictions) are based on prior knowledge and observations of the participants in other programming courses. Then, we perform a set of case studies to confirm whether they support the following predictions.

Hypothesis H1: Input/Output activity
Our first and instinctive prediction is that participants will mainly focused on input/output (I/O) activities by running the program rather than understanding the program itself.

Hypothesis H2: Visual scanning
Our second alternative prediction is that participants will be performing program comprehension tasks, such as scanning and reading source code, positioning eyes, indentations, and locating interesting parts of the source code to form a mental model. In addition, intentional naming and meaning of the identifiers in the source code can have an influence on recovering and extracting specifications closer to the application domain. All of these can have an influence on the comprehension and would help participants in understanding programs. Thus, participants form a mental model which is a representation of the program under investigation.

Hypothesis H3: Computer Science knowledge
Participants use a lot of computer science knowledge (*i.e.,* programming languages, diagramming, naming convention) and attempt to formulate some ideas about the source code starting from what they already knew. The topics covered in this computer science knowledge is closely related to students' computer science background.

Hypothesis H4: Application Domain
Students will try to map program entities and domain concepts which may lead to a better program comprehension. That is, participants have some prior knowledge at hand about the problem domain and consequently this reveals almost everything how the program works.

B. Participants

The participants were computer science students enrolled in software engineering and object-oriented programming courses. Most of the students in software engineering were undergraduates in the third or fourth year with good knowledge of Java. We refer to such a group of students as novices and their ages vary from 19 to 22. The other group of participants are graduate students enrolled in the object-oriented programming course and with two years of work experience. We refer to such a group of students as proficients and their ages vary from 22 to 26. This study was conducted practically by 48 students, 30 novices and 18 proficients. The group of proficients was substantially smaller than that of novices. There was no pretest to select participants in terms of levels of knowledge or programming experience. Each individual student aggregates a unit of result which in turn is analyzed within the corresponding case study.

C. Case Studies

To evaluate the effectiveness of our study, we have conducted a case study on three different Java programs where each case is centered around some object-oriented programming features.

Flesch-Kincaid:
This simple program is based on the Flesch-Kincaid readability test which is used to determine how difficult a passage of text is to be read by kids. For example, a score of 7.4 indicates that the text is understood by an average student in 7th grade. Details of the test are provided by Wikipedia at https://en.wikipedia.org/wiki/Flesch.

Pac-Man:
The second Pac-Man programming problem is of medium difficulty and it is a comprehensive exercise in the sense that the students are equipped with domain-general knowledge of the problem. This case study exposes occurrences of polymorphism, late binding, and object interactions. For details, refer to https://en.wikipedia.org/wiki/Pac-Man.

Othello:
The Othello complex problem (*i.e.,* inheritance, interfaces, GUI) is an online tragedy game often performed by professional and community theatre alike and where individuals can play this game against other players from around the world. It has been the source for numerous games, films, and literary adaptations. This case study exposes occurrences of polymorphism, late binding, and object interactions. For details, refer to https://en.wikipedia.org/wiki/Othello.

Table I shows the experiment containing three Java programs of different difficulty levels, easy, moderate, and difficult. In addition, the complexity of each program is reflected by the numbers of attributes, classes, and methods. For example, the difficulties in program Othello are expressive in terms polymorphic classes, GU, interfaces, and inheritance. This subsequently leads to difficulties in understanding the program by participants, in particular proficients.

Program Names			
Program's Attributes	Flesch-Kincaid	Pac-Man	Othello
Complexity Level	easy	moderate	difficult
Number of Classes and snippets	3 and 0	13 and 0	23 and 21
Number of Methods	6	34	48
Number of Statements	148	363	2164
Comments in Programs	no	partial	partial
Identifiers' Meanings	meaningless	meaningful, meaningless	meaningful, meaningless

TABLE I. Flesh-Kindcaid, Pac-Man, and Othello Java programs.

D. Data Collection

We started off each session by a short introduction of the study goals to the participants. We introduced program comprehension as the task of trying to understand a program (*i.e.,* source code) and how it works. We also emphasized on the rational behind the study and why we were interested in, that is, program comprehension is an important process of developing and maintaining software, and whether it should be broadly explored in the classroom. We also assured the participants of the anonymity and confidentiality of the data collected. In order to gain some insight about the study, we surveyed the participants using a questionnaire which has served as a guidance to explore whether the participants have any knowledge of program comprehension. At the time of the assessment, participants completed a short questionnaire

and were asked how much they known about the topic. The questionnaire consists of five questions stated as follows:

Note: Other similar terms for program comprehension are *"program understanding"* and *"source code comprehension"*.

1) I have never heard of program comprehension or similar terms and have no idea what they are.
   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree
2) In general, I know what program comprehension or similar terms are, but I have not used them.
   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree
3) To some extent, I have some familiarity with program comprehension or similar terms, but I have not used them.
   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree
4) I have used program comprehension or similar terms few times in other courses and projects.
   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree
5) I have often used program comprehension or similar terms in other curses and projects.
   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

The answers extracted from this questionnaire is all participants, novices and proficients, have never heard of program comprehension or similar terminologies. Therefore, the results constitute a representative audience for our study. The study was performed through two sessions, which took place at Northwest Missouri State University, and supervised by two faculty members and two teaching assistants. The experiment is conducted on laptops that have similar characteristics and with a fully configured NetBeans. Participants were not aware of the goal of the experiment in advance and were refrained from sharing information with other participants during the session. However, participant were briefly enlightened on the main idea behind the objective of the study. We did not impose nor provide any specific paradigm of comprehending the Java programs that we provided, and the participants are not forced to any specific approach.

IV. Results and Analysis

In each Java program, the participants started to understand the program through its execution traces, which was not an easy task in Othello because it is too complicated and large to be comprehended directly. The domain knowledge of the Pac-Man program was communicated through the names of identifiers, Blinky (B), Pinky(Y), Inky (I), or Clyde (C), which turns out to be ghosts from Pac-Man. Overall, identifiers are the most common source of information in all programs as they link the source code to the problem's domain. Both novices and proficients were able to comprehend the source code in just a few minutes. However, in the Flesch-Kincaid program, participants employed the input/output strategy, which is complemented by reading and visual scanning comprehension skills to further facilitate the comprehension process. Without any domain knowledge, Othello has remained a challenge for proficient students. However, in the short program, Flesch-Kincaid, there was no significant difference between novices and proficients with respect to comprehend the program in just a few minutes. Some novice students have prior knowledge and previous experiences implementing similar programs such as Othello. This has been shown in their performance (see Table II). In section III, we formulated several comprehension hypotheses before starting the experiment and the results have provided strong evidence for validating the hypotheses. Half of the participants in each group were not able to come up with all relevant test case scenarios. Thus, they were not able to establish a connection between the different parts of the program and different subsequent testing scenarios. We noticed that in a single program, some of the participants switched between multiple program comprehension hypotheses when faced with difficulties as we predicted.

In our study, we did not consider the age aspect of the participants. However, we focused on novices and proficients as the only indicators which differentiate between the participants. Table III shows a rubric of novices' and proficients' performances throughout Flesch-Kincaid, Pac-Man, and Othello programs. The novice's and profiency's ratings in this rubric showed that both Flesch-Kincaid and Pac-Man programs are suitable to participants' levels.

Program Names			
Techniques used (Hypotheses)	Flesch-Kincaid	Pac-Man	Othello
	Try/Succes	Try/Success	Try/Success
Input/Output Activities	63% vs. 95%	84% vs. 69%	35% vs. 48%
Visual Scanning	67% vs. 45%	34% vs. 88%	45% vs. 33%
Computer Science Knowledge	7% vs. 43%	64% vs. 10%	42% vs. 28%
Application Domain	12% vs. 23%	95% vs. 87%	56% vs. 63%

TABLE II. Novices' performance.

Program Names			
Techniques used (Hypotheses)	Flesch-Kincaid	Pac-Man	Othello
	Try/Success	Try/Success	Try/Success
Input/Output Activities	77% vs. 88%	90% vs. 55%	18% vs. 0%
Visual Scanning	53% vs. 65%	87% vs. 76%	2% vs. 0%
Computer Science Knowledge	31% vs. 56%	67% vs. 33%	12% vs. 4%
Application Domain	8% vs. 4%	94% vs. 87%	7% vs. 0%

TABLE III. Proficients' performance (Try/Success

The tuple (Try/Success) indicates the degree to which the program comprehension approach used attained a successful outcome. For example, 63% of novices used input/output activities (dynamic analysis) on the Flesh-Kindcaid program

and 95% were able to build a mental representation that relates the program's functionality to its source code. The greatest challenge for proficients is in the Othello program which involves inheritance and GUI.

In our case study, out of 48 participants, 30 were novices, and 18 were proficients. Table II and Table III show the summary of how participants performed in each Java program using different dimension of program comprehension. From Table III, the majority of the participants (84%) tried input/output activities on the Pac-Man program. Of these 84% of novices who tried the Pac-Man program using I/O activities, 69% of them succeeded. These I/O activities are similar to the black-box testing. In other words, participants selected an input scenario and observed the corresponding output. This activity helped participants to understand the overall functionality of the program. Since Pac-Man is a well-known game to participants, and the names of the identifiers are meaningful in the source code, computer science knowledge and domain knowledge helped novices succeed and able to execute the source code. Othello is one of the programs with a higher difficulty level, complicated source code, and with partial comments. Of all the three programs, participants were less successful in executing the program. However, novices comprehended the program better when compared to proficients. The reason is that novices had prior knowledge of the domain application. Therefore, domain knowledge and computer science knowledge played a vital role in comprehending the source code. In comprehending the Flesh-Kincaid source code, the majority of the participants tried and succeed by scanning the source code visually. The reason is the participants read the source code well and understand because the level of difficulty of the program is low.

The data gathered from this study gives insights about how participants followed their own comprehension instinct depending on several factors, which includes Java problems at hand, type of applications, and participants' skills. We noticed that both novices and proficients have performed I/O activities and visual scanning program comprehension activities in Flesh-Kincaid and Pac-Man programs. The results from our study as stated in Table II and Table III, validate our initial hypotheses, H1 and H2. However, for Othello, polymorphism which was difficult to debug has a negative impact on the participants, especially proficients. A possible justification is that novices were familiar to some extent with the Othello program while proficients have very limited knowledge about the application domain. Overall, the results showed that program comprehension activities depend on the problem context and there is no generic pattern to follow.

V. Conclusion

The goal of this paper aims at promoting program comprehension in software engineering and computer programming courses. We completed our work by exposing novice and proficient students to the concept of program comprehension, which is not part of the computer science or software engineering curricula. We ran the experiment with two groups of students and the results were promising, despite of the fact that students had no previous knowledge of program comprehension. In terms of correctness, the results turns out more convincing that domain knowledge and dynamic analysis of the source code are significantly important in comprehending programs. Overall, we attempt to pave the way and lay a background of program comprehension, an area of software engineering, that has not received much attention at both undergraduate and graduate levels and could be breathy investigated in various directions. We also expect to replicate this study with a higher number of participants and a larger code that could be refined to address the issues raised in this study.

References

[1] D. Erni A. Kuhn and O. Nierstrasz. Towards improving the mental model of software developers through cartographic visualization. *arXiv:1001.2386*.

[2] V. Murali B. Yuan and C. Jermaine. Abridging source code. In *Proceedings of the ACM on Programming Languages (OOPSLA 58:1)*, pages 13–20, ACM New York, NY, USA, 2017. ACM.

[3] Di Penta M. Oliveto R. et al. De Lucia, A. Labeling source code with information retrieval methods: an empirical study. *Empirical Softw. Eng.*, 19(5):1383–1420, 2004.

[4] Aziz Fellah and Ajay Bandi. Automata-based timed event program comprehension for real-time systems. In *Proceedings of FASSI 5th International Conference on Fundamentals and Advances in Software Systems Integration*, pages 21–28, Nice, France, 2019. IARIA.

[5] Aziz Fellah and Ajay Bandi. On architectural decay prediction in real-time software systems. In *Proceedings of ISCA 28th International Conference on Software Engineering and Data Engineering*, pages 98–100, San Diego, California, 2019. ISCA.

[6] Nathan Harris and Charmain Cilliers. A program beacon recognition tool. In *The 7th International Conference on Information Technology Based Higher Education and Training*, page 216225. IEEE, 2006.

[7] Yoshida N Kula RG Cruz AEC Fujiwara K Iida H Hongtanunam P, Yang X. Reda: a web-based visualization tool for analyzing modern code review dataset. In *International conference on software maintenance and evolution ICME"*, page 605608. IEEE, 2014.

[8] Janet Siegmund Ivonne Schrter, Jacob Krger and Thomas Leich. Comprehending studies on program comprehension. In *IEEE 25th International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, 2017.

[9] R. Ranca M. Allamanis M. Lapata J. Fowkes, P. Chanthirasegaran and C. Sutton. Tassal: autofolding for source code summarization. In *Proceedings of the 38th International Conference on Software Engineering Companion, (ICSE 16)*, pages 649–652, ACM New York, NY, USA, 2016. ACM.

[10] Thorsten Berger Thomas Leich Gunter Saake Jacob Kruge, Gul Calkl. Effects of explicit feature traceability on program comprehensi. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 338–349. ACM, 2019.

[11] M. Lanza and S. Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782795, 2003.

[12] Storey M. A. Theories, methods and tools in program comprehension: past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*, pages 181–191, Washington DC, DC, USA, 2005. IEEE Computer Society.

[13] Anneliese Von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[14] Harald Gall Michele Lanza, Stphane Ducasse and Martin Pinzger. Code crawler-an information visualization tool for program comprehension. In *In Proceedings of the 27th International Conference on Software Engineering*, page 672673. IEEE, 2005.

[15] Roper M. Pacione M.J. and Wood M. Code crawler-an information visualization tool for program comprehension. In *In Proceedings of the 27th International Conference on Software Engineering*, pages 70–79. IEEE, 2004.

[16] Ravindra Patel Rashmi Yadav and Abhay Kothari. Critical evaluation of reverse engineering tool imagix 4d. *SpringerPlus*.

[17] J. Siegmund. Program comprehension: past, present, and future. In *Proceedings of the 23th Internationa Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, pages 13–20. IEEE SANER, 2016.

[18] Margaret-Anne Storey. An interactive visualization environment for exploring java programs: Shrimp views revisited. In *Procedings of 9th International Conference on Program Comprehension*, page xviiixviii. IEEE, 2011.

[19] S. Xu. A cognitive model for program comprehension. In *Proceedings of the 3rd International Conference on Software Engineering Research, Management and Applications (ACIS '05)*, pages 392–398, Montréal, Canada, 2005.