Program Comprehension: A Short Literature Review

Matúš Sulír

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia

matus.sulir@tuke.sk

***Abstract*—Program comprehension is a study of how programmers understand existing programs. First, we delineate the research field and provide a brief overview of program comprehension theories. Then we present a literature review of program comprehension study results, techniques and tools – from overall comprehension through feature location to understanding the details and rationale behind the source code. Finally, future research directions are suggested.**

***Keywords*—debugging, feature location, mental model, program comprehension**

I. Introduction

According to Biggerstaff et al. [1], a person comprehends a program if one understands its structure, behavior and connection to the application domain. We can say that program comprehension describes how developers understand existing programs (whether or not written by themselves) in order to improve their functional or non-functional qualities.

A. Research Field Definition

To delineate the research field, let us put program comprehension in the context of a software development process (Fig. 1). As a developer reads the specification, a mental model forms in his or her head. A mental model is defined as a representation of a program in the programmer's mind [2]. This model is refined until the developer is able to implement parts of the application and create manually produced artifacts – traditionally the source code. The code is compiled, producing generated artifacts like a runnable application. To comprehend the program, the developer inspects the source code and debugs the produced application. This further refines the mental model and the development continues. Occasionally, the specification must be adapted because not all requirements were implemented in a desired way.

It is important to note that the order of the mentioned processes is not fixed and they often interleave or are performed in parallel. In many cases the programmer starts to familiarize with the program by inspecting generated artifacts – the running application. Finally, the diagram depicts only one specific developer's perspective and does not consider team aspects of software engineering.

B. Related Fields

Two most related research fields are software maintenance and reverse engineering. Maintenance is a much broader term which encompasses practically a whole software development process since it is difficult to distinguish between software creation and software modification.

The terms program comprehension and reverse engineering are often used interchangeably, although there is a slight difference. Reverse engineering focuses on techniques and tools for creation of higher-level abstraction documents from lower-level ones. Program comprehension research is also interested in the effect of using these tools. Therefore, reverse engineering techniques aid developers in program comprehension [3].

II. Theories and Methods

A cognitive model, also called a comprehension model, is a theory describing psychological processes involved in program comprehension which help to construct the mental model of a program [2]. There are multiple cognitive models described by researchers. Generally, we can divide them into top-down and bottom-up approaches.

Top-down program comprehension starts with the programmer's problem domain knowledge. The person forms hypotheses about the program and accepts or rejects them by inspecting the source code and other artifacts. The hypotheses are gradually refined into sub-hypotheses [4]. This type of comprehension is typically used when the program is familiar [5]. An example of a top-down cognitive model is the Brooks model [4].

Bottom-up comprehension is characterized by reading the source code line by line and incrementally grouping them to form higher-level abstractions [2]. An example is the Pennington model [6].

The fact that there are at least seven, relatively complex comprehension models described in literature (six in a review

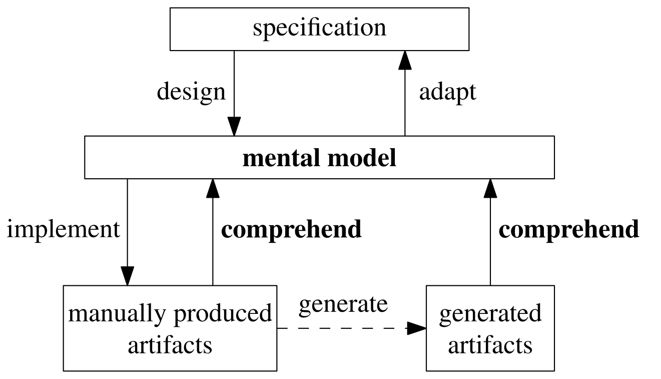

Fig. 1. Program comprehension in the context of a software development process. Solid arrows represent manual, human-performed processes; dashed arrows are automatic, computer-performed operations. The research area of program comprehension is marked in boldface.

[5] and one recent [7]) suggests us an idea that this number is not finite. Each person's cognition is quite different and it depends on a specific situation. It is therefore difficult to generalize the theories and design useful program comprehension tools directly after them. For this reason, many studies and experiments have been conducted in the field of program comprehension.

III. State of the Art

We will now explore the process of program comprehension from the most high-level perspective to detailed inspection, providing an overview of relevant techniques and tools available along with problems developers encounter.

A. Overall Comprehension

Architecture comprehension tools visualize the overall structure of a software system, its components and their relationship. Polymetric view tools like CodeCrawler [8] display code entities as nodes, relations between them as edges and arbitrary metrics as node shape attributes. For example, classes are displayed as rectangles with the width proportional to the method count and the height proportional to the line count. A disadvantage of such tools is the fact they usually focus just on simple quantitative metrics and perform only a static analysis of source code.

Metaphor-based visualizations transform code entities to 2D or 3D objects known from real life. For example, the famous city metaphor displays classes as buildings – like in the EvoSpaces tool [9]. An industrial application of such visualizations is questionable. A more promising approach is offered by ExplorViz [10], which combines them with so-called landscape view – a mixture of UML deployment and activity diagram produced by analyzing execution traces.

Domain analysis tools can be helpful for a new team member to become familiar with the system. The DEAL method [11] extracts a domain model from the GUI (graphical user interface) of a running application. However, manual user interaction is necessary and the method focuses only on a static structure of programs like relations between forms and their controls.

To gain the overview of domain knowledge contained in a program, it is possible to map program identifiers to an ontological dictionary like WordNet using subgraph homomorphism [12]. However, the approach is only semi-automatic and it is necessary to adjust the results manually. This is a result of representational defects like polysemy where one identifier in a program can represent multiple real-world concepts. Some types of defects can be detected and repaired, but the other are inherent to the nature of relationship between programs written in current general-purpose languages (GPLs) and the real world [13].

B. Location of Parts

As software becomes more complicated, it is impossible for a programmer to understand it whole. Therefore partial, or as-needed comprehension is necessary [14].

Each software consists of problem concepts and features (like a shopping cart or shipping method selection) and solution features (e.g., a web presentation and database persistence). One problem domain feature is scattered across multiple solution features. This is called feature delocalization [15]. One of the key objectives of program comprehension is to tackle this phenomenon which impedes code navigation by using feature location (or concept location) tools.

Textual code search is a common feature location approach. To find syntactical patterns, programmers often use complicated regular expressions and the results are unsatisfactory as programming languages are rarely regular. An AST (abstract syntax tree) querying is more viable. By using an intuitive "query-by-example" language [16], developers can search an AST without realizing it. However, the approach is not usable for more complicated patterns.

The MuTT tool [17] implements feature location by execution trace collection. While debugging the program, just before utilizing the feature of interest like adding an item to the shopping cart, the programmer clicks a button in the tool to start collecting the trace. After performing the desired action, the button is clicked again. A list of all methods executed between the two clicks is displayed in an IDE (integrated development editor). It must be manually filtered since it probably contains many auxiliary methods, not related to the tracked feature. Furthermore, to find multiple features, it is necessary to manually repeat the mentioned actions for each of them. The possibility of automation should be investigated.

After the features are located, it is desirable to mark their occurrences in the source code to avoid duplicate work in the future. This activity is called code labeling or concern tagging. In the previous example, we would mark all found methods with a tag "add to cart". An academic tool FLAT[3] [18] combines:

- full-text code search,
- feature location by execution trace collection using the MuTT tool,
- feature labeling with method-level granularity,
- and visualization of feature distribution across the source code files.

An another approach for feature location is differential code coverage [19]. If we run a program first selecting a feature of interest and then not, the difference of code coverage should be the source code of the feature. This approach suffers from similar limitations as execution trace collection.

It would be useful to combine execution trace collection of the MuTT tool with differential code coverage and assess the effectiveness of such approach. Furthermore, as bugs can be considered a kind of features (unwanted, of course), bug localization is a specific kind of feature localization. Differential code coverage could be used to localize the faults, too.

An interesting feature location tool is included directly in the core of an industrial IDE, NetBeans. During the runtime, it is possible to take so-called GUI Snapshot of the debugged program user interface and inspect the events to be performed after e.g., clicking a button or selecting a combo-box item.

C. Understanding the Details

The Theseus tool [20] shows how many times a particular method has been called during a web application execution. These numbers are displayed directly in the code editor and in real time. By clicking a given method, a retroactive log containing the argument values for each execution is shown. This allows for feature location along with a more thorough inspection of the program behavior. A similar tool [21], but for desktop Java applications, operates on individual source code lines instead of methods and thus focuses on detailed algorithm analysis.

Many difficult comprehension questions developers ask when debugging can be answered by asking reachability questions. A reachability question involves searching for all possible execution paths to find source code statements matching the given criteria [22]. However, developers' questions are often vague and difficult to formulate in mathematical terms. The overhead used to formulate such queries can exceed the benefits they provide.

As developers inspect the source code and associated artifacts using various techniques, they create mental models of the code which contain information far behind what is explicitly written. Despite there is a high demand for this implicit knowledge [23], it is not explicitly captured [24] or it is captured only in transient notes, not persisted across sessions [23]. One possible cause is that each developer's mental model is different and thus the personal notes created by one developer are not useful for an another one. This hypothesis should be tested by further research.

One form of temporary knowledge saving is the use of bookmarks in an IDE. To share such personal findings, collective bookmarks [25] can be used. In the study, they were found useful for marking the lines where to start with a given type of task, but not for detailed explanations.

D. Investigating the Rationale

Even if the code is thoroughly understood, programmers often ask why it was implemented this way [24]. One of the manually produced artifacts in Fig. 1 are version control system commits. Commit messages may contain invaluable information about intentions behind the source code.

Surprisingly, less developers actually read commit messages than write them [23]. A possible cause is that IDEs do not present them conveniently. Plugins like Deep Intellisense or Rationalizer [26] try to overcome this issue by displaying history information relevant to a given piece of code.

For version control analysis tools to be successful, each commit must contain code related to only one task. This is often not true in reality. A technique [27] can detect tangled changes before they are committed and suggest how they could be untangled. The disadvantage is a high ratio of false positives.

E. Making Programs Comprehensible

While analyzing existing software is useful, we must also learn lessons from it and design new systems to be more comprehensible.

There is an ongoing effort to create an intuitive programming language Quorum [28], designed after the results of empirical studies. However, it focuses too much on syntax of constructions (e.g., `repeat` instead of `for`) and copies the high-level design of semantics from the Java language.

Identifier naming is a significant factor of program comprehension. Method names can be evaluated for comprehensibility and more intuitive names can be suggested, using a database of commonly used n-grams [29].

Instead of using feature location approaches with often unsatisfactory results, there is a possibility to label classes and methods by their associated concerns using source code annotations immediately when writing the code. Then, source code projections [30] can be used to show classes and methods associated with a given concern in an IDE. While projections are perceived as useful, labeling the code manually is a time-consuming activity which does not directly affect the resulting program execution. Therefore, we can expect programmers to skip labeling or leave the annotations out of sync with the rest of the code, just as it often happens with comments.

Much work is done in the area of code clone detection and removal since there is a widespread opinion that duplicated code negatively affects comprehension. Surprisingly, a study [31] found out that fixing bugs affecting cloned code do not require more effort than fixing other bugs. Code with high regularity, which may be a result of cloning, is more easily comprehended by developers [32]. Furthermore, developers perceive code duplication in a broader sense than simple copy-pasting – for example, implementing the same method in multiple languages or in multiple version control branches [24].

IV. Future Directions

To design tools meeting the expectations of developers trying to understand complex systems, we must first perform experiments to study how they perform program comprehension using the tools available. Only then we can proceed to designing new ones. We plan to conduct qualitative studies designed to reveal the shortcomings of existing program comprehension approaches in specific contexts, produce hypotheses and then continue with quantitative experiments to accept or reject them.

A. Web and Mobile Applications

The majority of program comprehension studies and experiments are performed on a small or medium-sized project, usually self-contained and depending only on standard language and operating system API (application programming interface) and a few simple libraries, if any at all. The main area of interest are WIMP (windows, icons, menus, pointer) and command-line programs. Academic tools developed upon the results of these experiments are therefore usable only for projects subject to these limitations.

Nowadays, software projects are built using large frameworks. Systems have an immense number of dependencies and their behavior often depends on results of asynchronous calls to remote servers. They consist of multiple layers and having a web or mobile interface becomes a standard. Studying the comprehension of these systems is a promising research area.

B. Build Process Comprehension

Compilation and deployment processes of contemporary programs are sometimes more complex than the programs themselves. Especially when using generative programming and model-driven software development, it is important to know:

- what languages, frameworks, libraries and tools are used in the system,
- what are the dependencies between them,
- how do these artifacts interact to produce the intermediate results like object files and executables (if any)
- and how is the system configured to allow for the successful execution and presentation of results to a user.

C. Obtaining Terms from GUIs

In the source code, programmers are not technically obliged to use terms from the problem domain to name the identifiers. On the other hand, GUIs contain information shown directly to the end user an thus must contain domain terms. Using techniques like execution trace collection in combination with GUI ripping [33] could bring the terms back to the source code and at least partially substitute manual code labeling.

D. Language Composition

Many comprehension studies are performed on a program written in one general-purpose language. Today, language composition becomes a common way to construct programs – projects often use a mixture of severals GPLs and DSLs (domain-specific languages). The future research should investigate comprehension factors of using multiple languages in one program – not only from the syntactic aspect, but also the differences between individual paradigms and the effects of frequent switching from one paradigm to an another. It is important to know which combinations of languages and paradigms are more intuitive and closer to the mental model of the majority of programmers.

Acknowledgment

This work was supported by VEGA Grant No. 1/0341/13 Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.

References

[1] 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.

[2] M. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005, pp. 181–191.

[3] H. A. Müller and H. M. Kienle, "A small primer on software reverse engineering," University of Victoria, Tech. Rep., 2009.

[4] 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.

[5] A. von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, Aug 1995.

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

[7] J. Belmonte, P. Dugerdil, and A. Agrawal, "A three-layer model of source code comprehension," in *Proceedings of the 7th India Software Engineering Conference*, ser. ISEC '14. New York, NY, USA: ACM, 2014, pp. 10:1–10:10.

[8] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger, "CodeCrawler - an information visualization tool for program comprehension," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 672–673.

[9] S. Alam and P. Dugerdil, "EvoSpaces visualization tool: Exploring software architecture in 3D," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, Oct 2007, pp. 269–270.

[10] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live trace visualization for comprehending large software landscapes: The ExplorViz approach," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–4.

[11] M. Bačíková, J. Porubän, and D. Lakatoš, "Defining domain language of graphical user interfaces," in *2nd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), vol. 29. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 187–202.

[12] D. Ratiu and F. Deissenboeck, "Programs are knowledge bases," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 79–83.

[13] ——, "From reality to programs and (not quite) back again," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, June 2007, pp. 91–102.

[14] V. Rajlich and N. Wilde, "A retrospective view on: The role of concepts in program comprehension," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, June 2012, pp. 12–13.

[15] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[16] O. Panchenko, J. Karstens, H. Plattner, and A. Zeier, "Precise and scalable querying of syntactical source code patterns using sample code snippets and a database," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, June 2011, pp. 41–50.

[17] D. Liu and S. Xu, "MuTT: A multi-threaded tracer for Java programs," in *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, June 2009, pp. 949–954.

[18] T. Savage, M. Revelle, and D. Poshyvanyk, "FLAT^3: Feature location and textual tracing tool," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 255–258.

[19] K. D. Sherwood and G. C. Murphy, "Reducing code navigation effort with differential code coverage," Department of Computer Science, University of British Columbia, Tech. Rep., September 2008.

[20] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2481–2490.

[21] T. Matsumura, T. Ishio, Y. Kashima, and K. Inoue, "Repeatedly-executed-method viewer for efficient visualization of execution paths and states in Java," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 253–257.

[22] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 185–194.

[23] 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.

[24] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 492–501.

[25] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen, "Collective code bookmarks for program comprehension," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, June 2011, pp. 101–110.

[26] A. W. Bradley and G. C. Murphy, "Supporting software history exploration," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 193–202.

[27] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! Are you committing tangled changes?" in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 262–265.

[28] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *Trans. Comput. Educ.*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[29] T. Suzuki, K. Sakamoto, F. Ishikawa, and S. Honiden, "An approach for evaluating and suggesting method names using n-gram models," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 271–274.

[30] J. Porubän and M. Nosál', "Leveraging program comprehension with concern-oriented source code projections," in *3rd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), vol. 38. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 35–50.

[31] F. Rahman, C. Bird, and P. Devanbu, "Clones: what is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[32] A. Jbara and D. G. Feitelson, "On the effect of code regularity on comprehension," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 189–200.

[33] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, Nov 2003, pp. 260–269.