Annales Mathematicae et Informaticae
51 *(2020) pp. 103–111*
DOI: 10.33039/ami.2020.07.007
https://ami.uni-eszterhazy.hu

A comprehensive review on software comprehension models[*]

Anett Fekete, Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics
{hutche,gsd}@inf.elte.hu

Submitted: February 4, 2020
Accepted: July 9, 2020
Published online: July 23, 2020

Abstract

Software comprehension is one of the most important among software development tasks since most developers do not start a brand new software every time they switch jobs or get transferred from one project to another but join long-running software projects. Every experienced and expert developer has their own established methods of understanding complex software systems. These methods might be different for everyone but they still have common aspects by which multiple well-defined code comprehension models can be constructed. Furthermore, the degree of understanding of a software can be categorized as well, according to the ability of the programmer to modify or develop a certain part of the software system. This paper is intended to provide a review of the cognitive software comprehension models established by extensive research in this topic as well as describe the dimensions of understanding software. It also determines the editor support of cognition models by examining common editor functionalities and categorizing code editors based on the availability of functionalities of each cognition approach.

Keywords: code comprehension, comprehension model, code cognition, taxonomy

MSC: 68N99

[*]This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

1. Introduction

Since the very inception of computer science, software comprehension has been an ongoing challenge for everyone that ever tried to understand any unfamiliar code. Each programmer, even the ones with the least experience possesses some kind of – either conscious or subconscious – way of understanding source code. Naturally, more experienced developers have more sophisticated methods and workflows, since they are more familiar with the language, framework and architecture they work with.

Throughout the history of computer science, several researchers tried to grasp how a programmer approaches software comprehension and constructed high-level abstraction models from the collected information. There have been several empirical experiments done in this area where researchers observed the process of understanding unfamiliar source code and tried to draw comprehension patterns from the results. The acquired information was used to define mental models that determine a certain direction of thinking while a developer is executing code comprehension tasks. As a result of decades of research, several complete cognition models were constructed which can be classified as being one of two determinant approaches, or a combination of them.

Cognition models are easily applicable in everyday software development. If we consider code editor functionalities, we can see that they can be categorized into one or both approaches according to their individual purpose. Based on this classification, we can also determine which approach is supported by code editors.

Sec. 2 provides an overview of the common elements of comprehension models, followed by a comprehensive review on comprehension approaches illustrated by the most well-designed mental models of each category. We classify the common software comprehension features in standalone comprehension tools and code editors in Sec. 3 by the categories discussed in Sec. 2 and decide about the most popular editors which category they fit in based on their available features. Sec. 4 concludes our paper.

2. Models of code comprehension

Although computer science is a relatively new discipline, it has always been a great interest of software programmers to find a decent way to understand program code that has been written by fellow developers. One of the most frequent activities of a programmer is comprehending the code others wrote from the very beginning of their careers. Several prestigious companies tried to measure exactly how much time is spent looking at or searching for code: IBM found in 1989 that more than 50% of working hours are consumed by static analysis [3]. A research conducted by Microsoft in 2007 [2] showed that 95% of developers thought code comprehension was an important part of their daily tasks while 65% said that they engage in software comprehension activities at least once a day.

Every programmer, no matter how little experience they have, has their own, conscious or subconscious way of getting familiar with unknown source code. Young programmers at the beginning of their career often try to understand code in an ad hoc way, e.g. jumping from one part of the code to another while running the program. On the other hand, experienced programmers usually don't just hop into the middle of a code but tries understanding in a more structured, thus more effective way.

The methods applied by different programmers provide us with information about the workflow of software comprehension, from which several different structures can be extracted. A structure constitutes a mental model which is an abstraction of the software comprehension process. Several comprehension models have been constructed since the beginning of software production and multiple excellent research papers tried to collect and classify them based on similar viewpoints. Comprehensive research was done by von Mayrhauser [19] who also constructed their own mental model (see Sec. 2.4). Another similar review was done by Storey [18]. The paper of O'Brien [12] presents a somewhat different review as he compares the models based on their data collection methods. They all determined two main directions for software comprehension: top-down (see Sec. 2.2) and bottom-up (see Sec. 2.3).

As noticed by Levy [10], top-down models serve the purpose of learning about architecture and system components first, then move onto finer details. On the other hand, the bottom-up approach is the exact opposite, intended to obtain knowledge about smaller code snippets of a feature. However, the two directions can be switched between in an opportunistic way, thus creating combined comprehension models.

2.1. Common elements in comprehension models

All research on the subject revealed that comprehension models have common elements of which the models' components are built up. Practically, the elements are telltale code snippets and conjectures about the programming goals, and activities that brings the developer closer to the complete mental model.

- Static elements include recognizable patterns and clues in the source code as well as domain knowledge and conjectures.
  - Beacons [20] are signs standing close to human thinking that may give a hint for the programmer about the purpose of the examined code, e.g. function or variable identifiers.
  - Chunks [4] are coherent code snippets that describe some level of abstraction in the program (e.g. an algorithm).
  - Hypotheses [9] are assumptions about the source code based on domain knowledge that are a result of applying various comprehension techniques. They are classified by Letovsky [9] according to whether they

are aimed at the purpose (*why*), implementation method (*how*) or type (*what*) of a source code detail like a function.

  - Plans [17] include characteristic features of the source code that are so frequently used that they are easily recognizable.
    * Domain plans include high-level abstractions about the problem domain and its environment. It contains the mapping of real-world objects to programming objects (not necessarily meaning the objects of the object-oriented paradigm).
    * Programming plans describe typical practical concepts, e.g. data structures and their operations or significant details of algorithms.
  - Rules of programming discourse are the consensus about coding that are intended to facilitate comprehension by not having to adapt to other programmers' coding habits like coding style. Rules may determine coding conventions or data structure and algorithm implementations.
  - Text-structure knowledge [19] contains information about statements and commands in the source code and their relationships. It includes familiarity with control statement syntax and semantics.

- Dynamic elements are comprehension activities that bring the developer closer to the complete mental model.
  - Strategies include methods in the comprehension process to move from low-level abstractions to high-level ones.
    * Chunking [4] is the process of producing higher level chunks from lower level chunks. After repeating the process multiple times, high-level abstractions can be built.
    * Cross-referencing connects different abstraction levels by mapping the elements of source code to level description elements. Cross-referencing is the key step in building a mental model of the existing abstractions.

2.2. Top-down approach

Generally speaking, when applying a model that belongs to the set of top-down approach models means that the programmer starts the comprehension process from "the big picture" and gradually moves on to the smaller details of the project. The first step is to acquire a comprehensive system overview e.g. by running the program and placing breakpoints through which the programmer can trace the running process and locate the significant parts.

The developer in this case is usually equipped with some previous domain knowledge. In the theory of Brooks [1], comprehension is built upon the domain knowledge by constituting an initial hypothesis about the source code. This is later refined into follow-up hypotheses that are either proved right or wrong.

When the developer comes across a familiar algorithm, the same algorithm should be easily understandable for them in a different programming language or framework. This serves as a base to the cognitive model of Soloway, Adelson and Ehrlich [17], who also focus on the hierarchical structure of programming plans and goals. The plans are also ordered in their own hierarchical upbuilding. They say that programmers also make use of beacons and rules of programming discourse during the comprehension process.

2.3. Bottom-up approach

The bottom-up approach is the opposite of the top-down approach, as in when applied, programmers first try to understand the details of the code, then move towards the larger units by chunking the code statements. Shneiderman and Mayer [16] present a theory that consists of two main knowledge areas: the language dependent syntactic knowledge and the semantic knowledge that, although independent of any particular programming language, relies heavily on general programming knowledge. The semantic knowledge is built up of hierarchically structured layers from low-level details to the actual, high-level mental model.

Pennington [14] describes a similar, two-component model in her paper. However, unlike the previous model, the components here are rather coordinative than completing each other like the syntactic and semantic knowledge. According to Pennington, a program model is built first in the programmer's mind by observing the control-flow of the program. Then, a situation model is built while refining the program model in parallel, which incorporates the programming goals.

2.4. Combined approaches

Some cognition models apply both the top-down and bottom-up approach in some form; either they have a component that opportunistically applies one direction (or switch between them if needed) or utilize elements of other models from both approaches in their own components. An example for the former case is Letovsky's [9] model. Beside the knowledge base and the internal representation, it consists of a third component, the assimilation process which follows the discursive human thinking as it tries to acquire the most knowledge possible in the shortest possible time. During the assimilation process, the developer soaks up as much information as possible with the help of the knowledge base and external representations of the code (like documentation).

Another similarly high-level combined mental model was described by von Mayrhauser et al. [19], called the integrated metamodel. Four major components build up this model, two of them borrowed from Pennington's bottom-up model [14] (the program and situation model) and one borrowed from the top-down model of Soloway, Adelson and Ehrlich [17] (the top-down model). These three components are supported by a knowledge base. Any of the components can be activated at any time during the comprehension process.

3. Tool support for understanding the comprehension process

Other than domain knowledge like language syntax or coding conventions, programmers are aided by various software features during code comprehension activities. There are standalone tools that were made for specifically this purpose such as CodeCompass [15], CodeSurveyor [6] or OpenGrok [13]. These software provide a wide range of textual and/or visual information about the source code. However, most modern integrated development environments (IDEs) and code editors are also rich in code comprehension supporting functionalities [11, 18]. These can be categorized according to the cognition approach they support, top-down, bottom-up, or even both, when a functionality serves multiple actions in the comprehension process.

3.1. Editor functionalities

- **Call hierarchy views** support the top-down approach since they offer a well-structured view of the program's control-flow.
- **Code browsing**: top-down comprehension is intuitively helped by searching for previously captured beacons in the software files. On the other hand, control-flow and data-flow is also supported by code browsing which are key elements of bottom-up comprehension.[1]
- **Find all references** is an obvious tool for bottom-up comprehension since it serves as a navigation tool when the developer tries to get a hint of the usage of a symbol thus helps in chunking. Control-flow is also supported by this feature.
- **Go to definition** supports top-down comprehension because its main purpose is to find the definition (source) of a beacon thus helps the programmer move from higher to lower abstraction level.
- **Intelligent code completion** supports top-down comprehension as it offers the possibility to capture beacons by providing intuitive perspective of the various classes, functions and variables of a program.
- **Split view** provides top-down perspective as it makes the developer able to grasp beacons from multiple files at the same time. This functionality also supports typical bottom-up elements like data-centric views.
- **UML diagrams** are in support of top-down comprehension as their purpose is to provide a high-level visualization of the code structure (e.g. class diagram, activity diagram) and the program domain (e.g. use-case diagram).

[1]By code browsing we mean high-level navigation across files, classes, symbols etc., not a simple text search.

Functionality	Top-down	Bottom-up
Call graphs	✓	✗
Code browsing	✓	✓
Find all references	✗	✓
Go to definition	✓	✗
Code completion	✓	✗
Split view	✓	✓
UML diagrams	✓	✗

Table 1: Classification of editor functionalities based on cognition approaches

3.2. Editors and comprehension models

As Table 1 shows, most functionalities primarily support top-down comprehension. If we investigate the available tools in the most popular IDEs [11], we can determine which cognition approach is supported by a certain IDE.

	Top-down				Bottom-up	Both	
Editor	Call graphs	Go to definition	Code completion	UML diagrams	Find all references	Code browsing	Split view
Atom	✗	✓	✓	✗	✓	✗	✓
Eclipse	✓	✓	✓	✓	✓	✓	✓
JetBrains	✓	✓	✓	✓	✓	✓	✓
NetBeans	✓	✓	✓	✗	✓	✓	✓
Notepad++	✗	✗	✓	✗	✗	✗	✓
Sublime Text	✗	✓	✓	✗	✓	✓	✓
vim	✗	✓	✓	✗	✓	✓	✓
Visual Studio Code	✓	✓	✓	✗	✓	✓	✓

Table 2: Editor support of cognition approaches

Table 2 shows that the most popular IDEs and code editors support every software comprehension approach in general by providing the previously classified functionalities. All-purpose IDEs support most if not all examined features. Open-source IDEs like Eclipse and Visual Studio code are further extendable by software comprehension supporting plugins [5, 11]. It is also worth noticing that IDEs generally perform poorly regarding visual features such as call graphs or diagrams.

4. Conclusion

In this paper, we gave a comprehensive review on the various types of code comprehension models and their influence on software editors. We discussed their common elements, and categorized the cognition models based on their components and the direction of their workflow. We investigated several widespread code editor features and classified them considering whether they support a comprehension approach or not. This investigation allowed us to determine which of the most popular IDEs support the described approaches based on the availability of functionalities. We determined that the majority of IDEs support all approaches.

It is worth considering that well-defined cognition models seem to be overly strict regarding the human thinking process. Multiple research has shown that developers do not usually follow a rigorous pattern or a concrete model during their understanding activities, they rather apply opportunistic strategies [7]. For example, Koenemann et al. [8] concluded that programmers perform best in comprehension tasks when they try to understand only the relevant parts of the code in an as-needed manner. This means that cognition models work well as abstractions about the comprehension process and are also provide a good basis for IDE functionalities but they shouldn't be considered the only ways of 'correct' comprehension workflows.

References

[1] R. Brooks: *Towards a theory of the cognitive processes in computer programming*, International Journal of Man-Machine Studies 9.6 (1977), pp. 737–751,
doi: `https://doi.org/10.1016/S0020-7373(77)80039-4`.

[2] M. Cherubini, G. Venolia, R. DeLine, A. J. Ko: *Let's go to the whiteboard: how and why software developers use drawings*, in: Proceedings of the SIGCHI conference on Human factors in computing systems, 2007, pp. 557–566,
doi: `https://doi.org/10.1145/1240624.1240714`.

[3] T. A. Corbi: *Program understanding: Challenge for the 1990s*, IBM Systems Journal 28.2 (1989), pp. 294–306,
doi: `https://doi.org/10.1147/sj.282.0294`.

[4] J. S. Davis: *Chunks: A basis for complexity measurement*, Information Processing & Management 20.1-2 (1984), pp. 119–127,
doi: `https://doi.org/10.1016/0306-4573(84)90043-8`.

[5] L. Hattori, M. D'Ambros, M. Lanza, M. Lungu: *Software evolution comprehension: Replay to the rescue*, in: 2011 IEEE 19th International Conference on Program Comprehension, IEEE, 2011, pp. 161–170,
doi: `https://doi.org/10.1109/ICPC.2011.39`.

[6] N. Hawes, S. Marshall, C. Anslow: *Codesurveyor: Mapping large-scale software to aid in code comprehension*, in: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), IEEE, 2015, pp. 96–105,
doi: `https://doi.org/10.1109/VISSOFT.2015.7332419`.

[7] A. J. Ko, B. Uttl: *Individual differences in program comprehension strategies in unfamiliar programming systems*, in: 11th IEEE International Workshop on Program Comprehension, 2003. IEEE, 2003, pp. 175–184,
doi: https://doi.org/10.1109/WPC.2003.1199201.

[8] J. Koenemann, S. P. Robertson: *Expert problem solving strategies for program comprehension*, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 1991, pp. 125–130,
doi: https://doi.org/10.1145/108844.108863.

[9] S. Letovsky: *Cognitive processes in program comprehension*, Journal of Systems and software 7.4 (1987), pp. 325–339,
doi: https://doi.org/10.1016/0164-1212(87)90032-X.

[10] O. Levy, D. G. Feitelson: *Understanding large-scale software: a hierarchical view*, in: Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, 2019, pp. 283–293,
doi: https://doi.org/10.1109/ICPC.2019.00047.

[11] M. Mészáros, M. Cserép, A. Fekete: *Delivering comprehension features into source code editors through LSP*, in: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2019, pp. 1581–1586,
doi: https://doi.org/10.23919/MIPRO.2019.8756695.

[12] M. P. O'brien: *Software comprehension–a review & research direction*, Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report (2003).

[13] Oracle: *OpenGrok: "A wicked fast source browser"*,
url: https://oracle.github.io/opengrok/.

[14] N. Pennington: *Stimulus structures and mental representations in expert comprehension of computer programs*, Cognitive psychology 19.3 (1987), pp. 295–341,
doi: https://doi.org/10.1016/0010-0285(87)90007-7.

[15] Z. Porkoláb, T. Brunner, D. Krupp, M. Csordás: *Codecompass: an open software comprehension framework for industrial usage*, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 361–369,
doi: https://doi.org/10.1145/3196321.3197546.

[16] B. Shneiderman, R. Mayer: *Syntactic/semantic interactions in programmer behavior: A model and experimental results*, International Journal of Computer & Information Sciences 8.3 (1979), pp. 219–238,
doi: https://doi.org/10.1007/BF00977789.

[17] E. Soloway, K. Ehrlich: *Empirical studies of programming knowledge*, IEEE Transactions on software engineering 5 (1984), pp. 595–609,
doi: https://doi.org/10.1109/TSE.1984.5010283.

[18] M.-A. Storey: *Theories, methods and tools in program comprehension: Past, present and future*, in: 13th International Workshop on Program Comprehension (IWPC'05), IEEE, 2005, pp. 181–191,
doi: https://doi.org/10.1109/WPC.2005.38.

[19] A. Von Mayrhauser, A. M. Vans: *Program comprehension during software maintenance and evolution*, Computer 28.8 (1995), pp. 44–55,
doi: https://doi.org/10.1109/2.402076.

[20] S. Wiedenbeck: *Beacons in computer program comprehension*, International Journal of Man-Machine Studies 25.6 (1986), pp. 697–709,
doi: https://doi.org/10.1016/S0020-7373(86)80083-9.