

Intensions are a Key to Program Comprehension

Václav Rajlich

Department of Computer Science

Wayne State University

Detroit, MI USA 48202

rajlich@wayne.edu

Abstract

The classical comprehension theories study relations between extensions, intensions, and names. Originally developed in linguistics and mathematics, these theories are applicable to program comprehension as well. While extensions are present in the program, the intensions are usually missing, and evolution and maintenance programmers have to recover them as the program cannot be successfully comprehended and changed without them.

There are six fundamental processes of comprehension and they have several attributes that generate a large set of comprehension processes. One of these processes is concept location, which is a practical and theoretically interesting problem of program comprehension.

Despite the current divergence among program comprehension processes, there is still a possibility that a universal program comprehension process will emerge sometime in the future.

1. Introduction

Software development presents numerous challenges. The lesser among them are the accidental difficulties that are tied to the specific technologies and processes that software engineers employ in their project. Examples of such accidental difficulties are compiler bugs, quirks in the programming languages, technology gaps that make certain tasks more difficult than they should be, ill thought-out processes that require unnecessary steps and ignore necessary tasks, and so forth. The accidental difficulties share a common trait: they are limited to a specific project's circumstances and they are usually resolved in a due time.

A different situation arises when dealing with essential difficulties. They are a subset of the essential properties of software and as long as software exists

they will cause problems. The essential properties were studied by Fred Brooks [3], who identified four essential difficulties: invisibility, complexity, conformity, and changeability. All four, each in its own peculiar way, complicate program comprehension.

Invisibility means that the role of the senses in program comprehension is limited. Since we often use our senses in comprehension, this makes the comprehension hard. This difficulty can be alleviated by various visualizations and sonifications, but in order for these approaches to be practical, considerable work is still required [21].

Complexity is another essential software difficulty. Programs nowadays consist of millions of lines of code and hence their complexity is comparable to the other complex systems created by humans, like large cities. Our short-term memory can accommodate only about 7 concepts at any given time [12]. In consequence, we have to employ various aids and strategies.

Changeability means that programs are easy to change; note that changing programs *correctly* may be much harder. Nevertheless, as a consequence of easy changeability, programs change frequently and yesterday's comprehension may be obsolete today.

Conformity means that programs interact with and hold together large systems that include hardware, users, and domains. These systems are reflected in the programs in various ways. For example Point of Sale program must reflect the hardware it runs on, users (store clerks) and their capabilities, the domain including intricacies of the credit card payment, and so forth. This of course adds even more complexity.

Program comprehension is a programmers' struggle with these essential difficulties and ICPC proceedings chronicle the successes and setbacks of the program comprehension efforts. On the positive note, the progress of program comprehension is the progress on perhaps the most important software engineering front, the fight against essential software difficulties, and

every important result represents a lasting progress for this field.

The rest of the paper is organized in the following way: Section 2 surveys the classical comprehension, section 3 discusses the attributes of program comprehension processes, section 4 presents concept location as an example of a comprehension process, and section 5 contains conclusions and future work.

2. Classical comprehension

Software engineers are not the first people who had to cope with a comprehension problem; before them, at the turn of the 19th and 20th century, mathematicians and linguists faced a similar problem. Both mathematics and linguistics also had to deal with some of the same essential difficulties: complexity and invisibility are present in both of them. Conformity is an issue in both applied mathematics and linguistics; however, pure mathematics decided not to deal with it and limits its attention to abstract systems only. Changeability exists in both mathematics and linguistics (new notations, definitions, and theorems are introduced, the languages under investigation change), but is much slower; therefore, the classical comprehension theories do not address this essential difficulty.

Since these fields share some of the software essential difficulties, their comprehension problem is related to program comprehension. Their comprehension questions can be summarized as “What does this sentence mean?” or “What does this mathematical formula mean?”

In linguistics, De Saussure (1857 – 1913) [7] introduced the following notions that comprise comprehension: There is *the signifier*, which is the sound or string of letters that the person recognizes. There is *the signified*, which is the idea or the concept that the signifier denotes. There is *the referent*, which is the actual thing or object that the signifier refers to.

In mathematics, Gottlob Frege (1848 – 1925) [8] introduced the following notions: *formula* is a sequence of symbols, *intension* is a complete set of meanings or properties, *extension* is a set of all things intension applies to.

In the field of artificial intelligence, several projects, roughly from 1972 [22] through 1992 [9], attempted to aid program comprehension. “Program understander” [23] of Figure 1 summarizes a shared, but never fully attained, goal of these projects. It analyzes source code and creates an internal program representation. There is a plan library and each of the plans corresponds to an implementation of a concept. Recognizer matches these

plans against the program representation, producing recognized concepts that aid program comprehension.

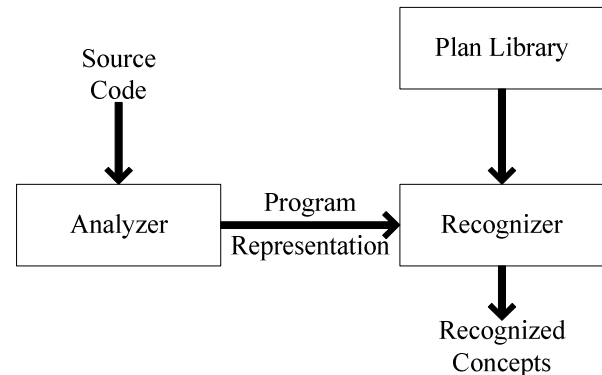


Figure 1. Goal of past artificial intelligence projects: Program understander.

Different terminologies by different authors can create confusion and are summarized in Table 1. The columns belong to the specific authors or fields. The terms in each row are in this paper considered synonyms; the differences in their meaning are minor. In order to make reading easier, each row of Table 1 has one term highlighted in bold and the rest of the paper uses this term as a substitute for the remaining terms in their respective rows. Moreover the paper will deviate from the common usage of term “concept”, using it in a broader sense of a label for the whole triple of name, intension, and extension.

Table 1. Terminology of comprehension.

| De Saussure | Frege | AI | ICPC |
|-------------|------------------|----------------|---------------------------|
| signifier | formula | label | name |
| signified | intension | <i>concept</i> | concept \supset feature |
| referent | extension | plan | implementation |

As an example of a concept, consider concept name “dog”, or “pes” in Czech, or “Hund” in German. The corresponding intension is “A hairy animal with four legs and teeth” (this intension is obviously incomplete and needs additional work). The corresponding extensions are real dogs like Fido or Lajka, pictures of dogs in books, magazines, movies, cartoons, and also Buck of Jack London’s book “Call of the Wild” who may have existed only in Jack London’s imagination and later in the book.

2.1. Concept triangle

Classical comprehension can be summarized by the concept triangle of Figure 2; the whole triangle represents a “concept” and has vertices “name”, “intension”, and “extension”. Comprehension of a concept means that the comprehending person is able to move among all three vertices of the triangle.

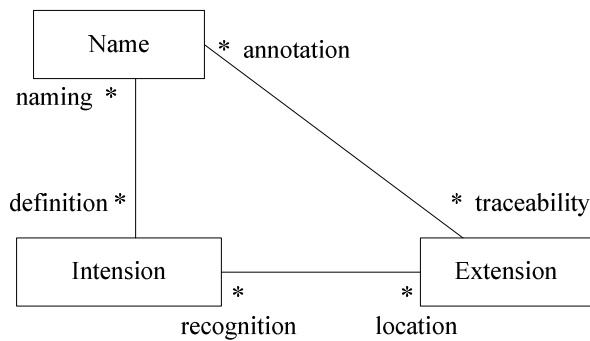


Figure 2. Classical concept triangle.

There are six fundamental comprehension processes. *Naming* gives a name to an intension. The opposite process is *definition* that for a given name finds the corresponding intension. The relation between names and intensions is many-to-many and, therefore, we have homonyms and synonyms that the naming and definition processes have to deal with. There may be additional difficulties with definition process that were pointed out in [17, 26].

Recognition recognizes in an extension a corresponding intension. The opposite of recognition is *location* that finds for a given intension a corresponding extension. Concept location is an important part of software change [2, 25]. Before the program change is attempted, the appropriate extension must be found. Concept location is practiced by programmers on daily basis and goes hand-in-hand with the change request.

Finally, there is *annotation* that recognizes an extension and gives it a name. The opposite is *traceability* that for a given name finds corresponding extensions. Already mentioned program understander of Figure 1 was a tool that was supposed to do annotations automatically, by recognizing extensions and giving them names. The opposite process, traceability, is frequently used in books, where a book index allows the reader to move from a concept name to pages where the concept extensions are discussed.

3. Attributes of program comprehension

The classical comprehension and its six fundamental processes provide a solid foundation for the program comprehension. However, program comprehension has developed additional insights that clarify attributes of the six basic comprehension processes.

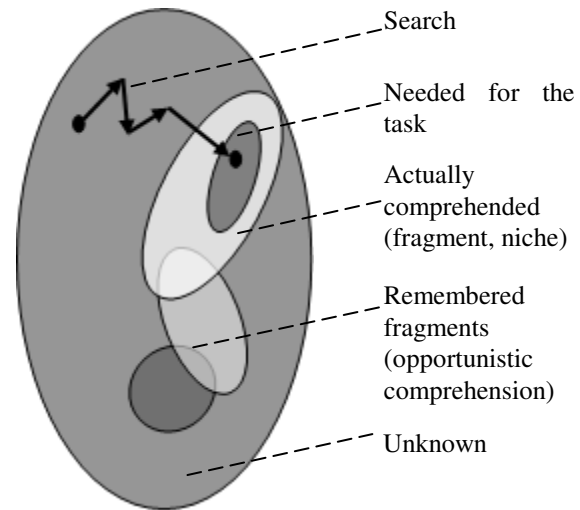


Figure 3. As-needed comprehension.

3.1. The scale

The first such attribute is the *scale* of the comprehension. Classical approaches tacitly assumed *whole-scale* comprehension where the whole system must be comprehended. However, this whole-scale comprehension is unrealistic and the *as-needed* approach is necessary [10]. In such situation, the programmers attempt to comprehend only a part of the system (*fragment, niche*) and the rest of the system remains unknown.

In as-needed comprehension, the programmers must perform various searches through the unknown part of the system, looking for the desired concept; in large systems, the process can be like the proverbial search for the needle in a haystack, see Figure 3.

Programmers are interested in the fragment of the current interest, but often they are also interested in fragments that were comprehended during previous tasks and this past comprehension is either remembered or preserved in the documentation [18]; this variant of the comprehension process is called *opportunistic* comprehension.

Because of the frequent searches, as-needed comprehension, deals with the whole system rather than isolated concepts. The name is treated as a part of

vocabulary, intension as a part of ontology, and extension as a part of program or a larger system that the program interacts with. Vocabulary, ontology, and program/system constitute the new vertices of the comprehension triangle. They are comprehended only partially and, hence, have known and unknown parts; comprehension processes in the known parts are different from comprehension processes in the unknown parts. The modified comprehension triangle is in Figure 4.

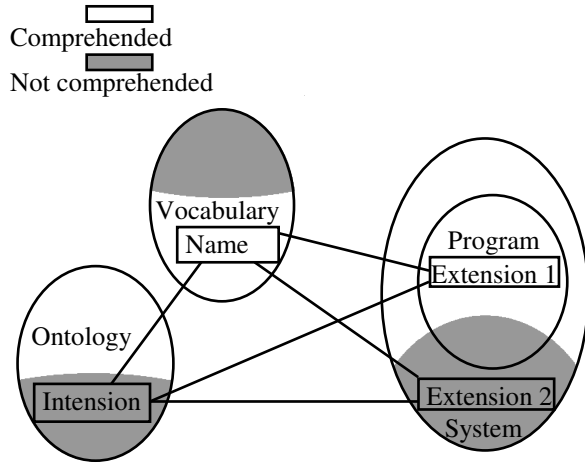


Figure 4. Comprehension triangle for opportunistic comprehension.

3.2. The speed

Another attribute of the comprehension processes is the *speed*. Classical comprehension does not address this attribute but tacitly assumes a *swift* and instantaneous process. However the processes of comprehension, particularly in the unknown part of the system, may be *slow*, consisting of many steps, and may involve searches and backtracks. More recent cognitive theories of program comprehension recognized this arduous nature of program comprehension [5, 13, 20, 24].

One of the factors contributing to this slowness is the many-to-many nature of the relations in the classical comprehension triangle. For example, as mentioned earlier, the conformity of the programs means that the programs interact with the problem domain, users, hardware, other programs, and so forth; as the result, the extensions of a specific concept can be often found in several parts of the system, as illustrated in Figure 5. The programmer may search for all these parts or only some of them; it may not be immediately apparent which parts are essential when trying to comprehend

the concept. Moreover, the program itself often consists of several tiers: there can be a processing tier, a data tier, and a graphic user interface, and the extensions of the same concept can appear in one or several of these tiers.

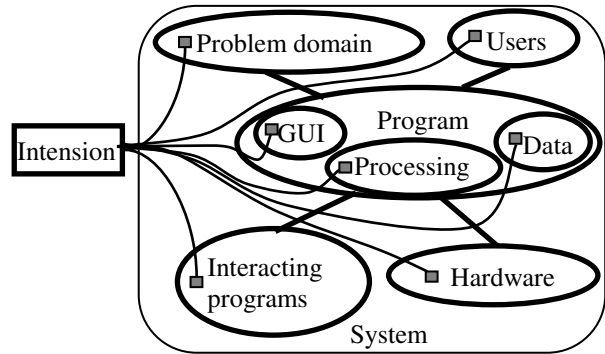


Figure 5. Many extensions in the system.

For another example, suppose there is the following statement in the code:

```
ix *= 1.06
```

The value 1.06 can mean many different things. The intension can be, for example, the Michigan sales tax, as Michigan levies 6% sales tax on all non-food purchases. In this case, the variable *ix* stores the amount to pay. The intension could be also the factor by which gas consumption increases when driving fast and, in this case, variable *ix* stores the amount of gas necessary for a trip. It can also mean a number of other things and there is no way to tell the corresponding intension from this isolated statement. This problem of mapping extensions to the appropriate intensions is one of the reasons why the goals of the program understander were never fully attained.

In a well written code the identifiers, comments, and documentation play an indispensable role: they preserve the concept intensions or concept names and hence make the extensions comprehensible. When searches are performed for the comprehension of a software system, the information stored in the identifiers, comments, and documentation is often used in queries. However, even in this case inaccuracies, omissions, homonyms, and synonyms can lead to a situation where the exact query may not lead to any results, and programmers would know only retroactively whether they found what they were looking for. The same situation can happen when looking for a name in the vocabulary, or for an intension in the ontology. The searches done during comprehension follow the style outlined in [11] and are represented by the activity diagram in Figure 6.

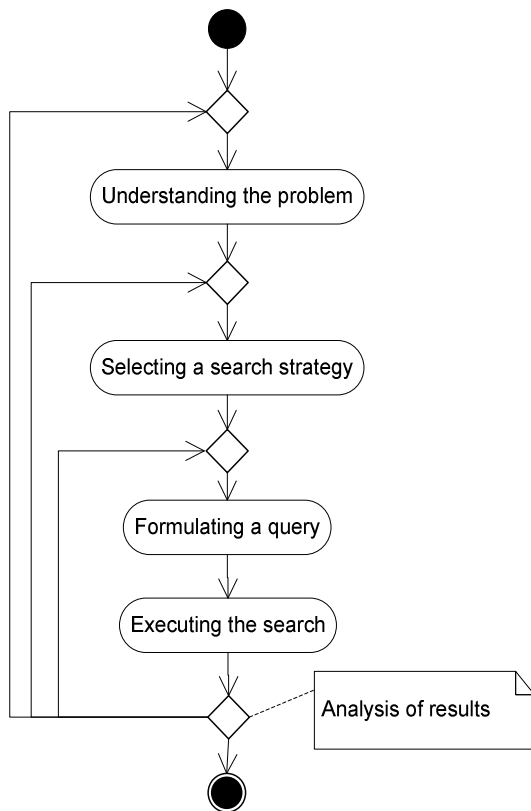


Figure 6. Searches during comprehension.

3.3. The levels of automation

In classical comprehension, all the processes are performed by humans. Artificial intelligence projects opt for the opposite extreme: everything is attempted to be done by the computer. Current approaches lie somewhere in between, i.e., some tasks are done by humans, other are assumed by the computer.

The philosophy of this approach was named by Brooks “intelligent assistance” [4] and calls for the computer to take responsibility for the algorithmic tasks that it does the best, and the humans to assume the tasks that require informal judgment that they do the best. Figure 7 contains an example of an interactive comprehension process; it contains a UML activity diagram [1] with two “swimlanes”, one for the computer and one for the human programmer. The control passes between both swimlanes. The specific process in Figure 7 is concept location by dependency search, discussed in more details in section 4.

3.4. Other attributes

Besides the three attributes of scale, speed, and automation, there are additional important attributes of program comprehension processes. There is a *level of detail* in comprehension, where the actors may gloss over things that they deem unimportant and concentrate only on a simplified set of issues. This attribute is addressed in the discussions of models and modeling [19].

Another attribute is related to the *properties of the system*. The processes are different if the system contains clues like identifiers, comments, or documentation, or if the system does not give these clues. Programmers also have to deal with both existing systems and systems that are under construction.

The *properties of the extensions* also influence comprehension. Some extensions can be *explicit* in the code, i.e., it is possible to point to program statements and fields that constitute these extensions. Other concepts are *implicit*, i.e., there is only an indirect indication of their presence [16]; they are implied by the code, but not explicitly expressed. For example, in basic word processors an implicit concept would be the authorization to open files; any user is authorized to access any word processor file. This authorization cannot be directly located since it is not explicitly implemented in the code, it is present as an assumption that underlies in certain parts of the code. Determining the location of such implicit concepts represents a special challenge.

Another property is the *evolution* of the system, ontology, and vocabulary over the time. This evolution occurs due to the volatility of the programs and systems, and also due to a programmer learning, which causes the evolution of both the ontology and the vocabulary [27]. The volatility of all three vertices of the comprehension triangle favors as-needed comprehension and results in slow processes.

In summary, there are six classical processes of comprehension, each with several attributes, where each of these attributes can assume a number of different characteristics. This results in a huge number of various comprehension processes and offers the ICPC community numerous interesting topics to explore for many years to come.

4. Concept location as an example

Concept location is the first activity of software change. During concept location the programmers find the location of the code that the software change will modify. In terms of fundamental comprehension

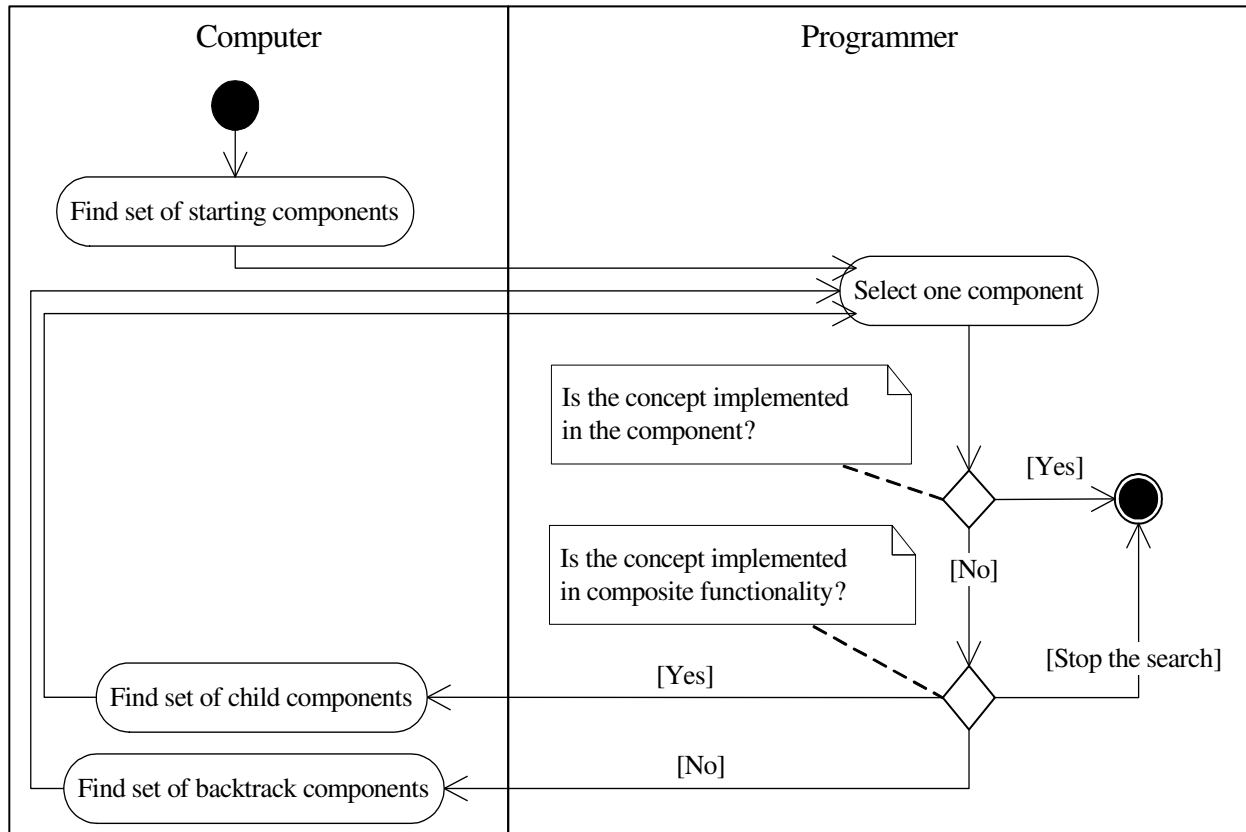


Figure 7. Activity diagram of concept location by dependency search.

processes, concept location starts with an intension and finds the corresponding extension. In a well documented code with good identifier names, concept location often takes a detour, first from concept intension to concept name and then from name to concept extension.

The process of concept location has the following attributes: It is an as-needed process; it does not try to create whole-scale comprehension, but rather tries to find and comprehend a part of code containing concept extension. It is also a slow process, with many steps and possible backtracks. It is currently practiced as an interactive process where the tool analyzes the program and programmer makes the relevant decisions.

The modeling techniques have not been found useful during concept location; perhaps, the reason is that current insights do not allow to separate important and unimportant aspects of the system, so the programmers opt to deal with the system as-is, rather than loosing details that may turn up to be decisive.

Many concept location techniques depend on the existence of the clues in the code, including identifiers and comments. Locating concepts in the code that does

not contain such clues is a very different problem that has been explored much less.

Programmer learning during concept location was documented in [14]; as the programmers search through the code for the appropriate concept extension, they learn more about the related concepts and the code, and leverage this increased knowledge in the search.

4.1. Naming: From concept intension to concept name

The programmers identify the concept intensions that are at the core of the change request. In many instances, these intensions use the relevant concept names directly; in that case, the concept name appears in the change request as a noun, a verb, or a clause. The change request can have many such nouns, verbs, or clauses and the programmers must extract the significant concepts, i.e., the ones that will lead to the correct location in the code where the change will be implemented.

If there are several concepts that can be gleaned from the change request, the programmers must asses

which one is the most significant one and most likely to be found in the code and which ones are crucial for the change. One way how to find significant concepts is to follow the following process:

- analyze the change request;
- extract the set of concepts used in the change request;
- delete the concepts that are intended for the communication with the programmer and do not deal with the new functionality;
- delete the concepts that are unlikely to be implemented in the code, like concepts related to the things that are outside of the scope of the program;
- organize the remaining concepts by the likelihood of being easy to located in the code.

As an example, suppose that the programmers deal with Point of Sale system and the change request is “Implement a credit card payment”. In this change request, we can identify the following concepts: “implement”, “credit card”, and “payment”.

Concept “implement”, although a part of change request, is not the significant concept, because it is a generic term that commands the programmer and is unrelated to the new functionality; there is no specific location in the code that would deal with it and therefore it can be excluded.

Concept “credit card” describes a functionality that is not in the old code but must be introduced in the future; therefore, it is not a significant concept that could guide to the location in the code.

The only remaining concept at this point is “payment”; “payment” is the significant concept that has to be located in the code before the change can start. Payment is very likely to be implemented somewhere in the code, most probably as a variable. The programmers must find this variable in order to start the change.

Once the extension of this concept is located, it can be expanded by the software change into the full payment functionality that also allows the payment by credit card.

4.2. Concept location by dependency search

There are many concept location techniques available to the programmers; for example, a recent survey can be found in [15]. In this paper, we briefly survey two techniques: Concept location by dependency search, and “grep” technique.

Concept location by dependency search [6] is represented by the activity diagram of Figure 7. The search is based on the following observations: Each

program component contains certain concept extensions, for example, there is a class Register that contains extension of concept “payment”. All concepts that are located in the class Register are called local functionality of Register. The purpose of the dependency search for concept “payment” is to find the component that contains extension of “payment” in its local functionality.

The search is guided by another kind of functionality called composite functionality. Composite functionality goes beyond the local functionality and it is the functionality that the class delivers to its clients. It includes not only the concept extensions located in class Register, but also all concept extensions located in all supporting classes.

On the top of the class hierarchy is the top class that is responsible for the functionality of the whole program and delivers it to the user. All concept extensions of the program are part of the composite functionality of the top class. However, the top class does not implement all these concept extensions, but rather delegates most of them to the other classes. These other classes in turn delegate them further down to the classes that are lower in the hierarchy. This fact is used by programmers who search for concepts, and recognize presence of the concept extensions in both local and composite functionality.

The search starts with computer action “Find set of starting components”, see Figure 7. In this action, the computer identifies the top classes in the program. Then the programmer selects one of them and decides whether the concept is implemented in the local functionality. If yes, the class is the location of the concept and the search is done.

If the concept is not found in the local functionality, the programmer has to determine whether the concept is implemented in the composite functionality. For that, the programmer uses various clues like the identifiers, comments, and so forth. Usually, there is no need to read the code of the whole set of supporting classes, it is sufficient to make a rough guess. If the guess turns out to be wrong, it will lead to a later backtrack; this backtrack does not invalidate the search, only makes is a little bit longer.

If the programmer concludes that the concept is implemented in the composite functionality, then the computer finds the set of child classes. The programmer selects the most likely one among them and the search continues with another iteration.

If, on the other hand, the programmer concludes that the concept is not present in the composite functionality, it means that a wrong turn was taken sometimes before. The programmer then must

backtrack to a previously visited class and visit a different child then before. The computer keeps track of all visited classes, and the programmer chooses one of them to backtrack into. If the search appears fruitless, the programmer can stop it without locating the concept, perhaps with the idea that there should be a search for a different concept or a different search technique should be used.

4.3. Concept location by grep

Searching by grep is a popular concept location technique. It tries to find in the code the concept name or a similar text [14], assuming that the use of the concept name in the code indicates a presence of the concept extension. The programmers formulate regular expression queries (“patterns”) and query the source code of a software system using grep tool. For example when looking for the concept “payment”, programmers may query for the original word (payment) or various abbreviations and variants (pmt, paymt, pay_1, and so forth.)

The tool grep produces a list of code lines where the desired name appears, called “matches” or “hits”. There may be several matches and the programmers read the code that surrounds these matches. Based on this reading, they decide whether the concept has been located. The query fails when the set of matches is empty; this indicates that the sought name (or its chosen variant) is not used in the code. Another situation arises when none of the matches contain the needed extension, because the word is used in the code with a different meaning. In both of these cases, the programmers have to do a new search.

Sometimes, the query produces too many matches and it is not practical to inspect them. In this situation, the programmers formulate an additional query and do another search, this time searching only the set of the earlier matches. In this way, they get a set-theoretical intersection of the results of the two queries and this resulting set of matches should be smaller than the earlier large set.

5. Conclusions and future work

In this paper, we surveyed the field of program comprehension, both classical and current. The comprehension triangle of intension, extension, and name is the foundation of the comprehension efforts; there are six fundamental comprehension processes in this triangle: naming, definition, recognition, location, annotation, and traceability. These processes have the attributes of scale (whole-scale vs. as-needed), speed

(swift vs. slow), automation (human vs. fully automatic vs. interactive), level of detail (as-is vs. model), property of system (annotated vs. not annotated, existing vs. being created), properties of extensions (explicit vs. implicit), and evolution (static vs. volatile). The basic processes together with the attributes generate a large number of processes to be investigated by comprehension researchers.

Current state-of-the-art in program comprehension is characterized by a large number of the comprehension processes. In spite of this wide variety, the comprehension processes still may converge in the future into one or few processes that would be fully automatic, whole-scale, and swift, possessing attributes that the classic approaches and “program understander” assumed or dreamed of but never attained. However, path to such processes is still long.

People who deal with other complex systems, like biological systems, also face comprehension problems. The comprehension triangle and six fundamental processes are very likely universal and, therefore, applicable to other systems also; however, process attributes of other systems may be very different. Nevertheless, the accumulated program comprehension solutions may contribute to the comprehension processes within these other systems also.

Acknowledgements

The author would like to thank Radu Vanciu and Maksym Petrenko for their help in preparation of this paper. This work was partially supported by the grants CCF-0438970 and CCF-0820133 from the National Science Foundation (NSF). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

References

- [1] Arlow, J. and Neustadt, I., *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley Professional, 2005.
- [2] Biggerstaff, T. J., Mitbender, B. G., and Webster, D., "The Concept Assignment Problem in Program Understanding", in *Proceedings International Conference on Software Engineering (ICSE 1993)*, Baltimore, Maryland, 1993, pp. 482 - 498.
- [3] Brooks, F. P., "No silver bullet: Essence and accidents of software engineering", *IEEE Computer*, 20, 4, April 1987, pp. 10-19.

- [4] Brooks, F. P., "The computer scientist as toolsmith II", *Communications of the ACM*, 39, 3, March 1996, pp. 61-68.
- [5] Brooks, R., "Towards a theory of the cognitive processes in computer programming", *International Journal of Man-Machine Studies*, 9, 6, 1977, pp. 737-742.
- [6] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in *Proceedings 8th IEEE International Workshop on Program Comprehension (IWPC 2000)*, Limerick, Ireland, June 2000, pp. 241-249.
- [7] de Saussure, F., *Writings in general linguistics*, Oxford University Press, 2006.
- [8] Frege, G., *The Basic Laws of Arithmetic; Exposition Of The System*, Berkeley: University of California Press, 1964.
- [9] Kozaczynski, W., Ning, J., and Engberts, A., "Program concept recognition and transformation", *IEEE Transactions on Software Engineering*, 18, 12, 1992, pp. 1065-1075.
- [10] Lakhota, A., "Understanding someone else's code: Analysis of experiences", *Journal of Systems and Software*, 23, 3, 1993, pp. 269-275.
- [11] Marchionini, G., *Information Seeking in Electronic Environments*, Cambridge University Press, 1997.
- [12] Miller, G. A., "The magical number seven, plus or minus two: Some limits on our capacity for processing information", *Psychological Review*, 63, 8, 1956, pp. 1-97.
- [13] Pennington, N., "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", *Cognitive Psychology*, 19, 1987, pp. 295-341.
- [14] Petrenko, M., Rajlich, V., and Vanciu, R., "Partial Domain Comprehension in Software Evolution and Maintenance", in *International Conference on Program Comprehension (ICPC 2008)*. Amsterdam, Holland, 2008, pp. 13-22.
- [15] Poshyvanyk, D., Guéhéneuc, Y., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, 33, 6, 2007, pp. 420-432.
- [16] Rajlich, V., Gosavi, P., "Incremental Change in Object-Oriented Programming", *IEEE Software*, 21, July/August, 2004, pp. 62 - 69.
- [17] Rajlich, V. and Wilde, N., "The Role of Concepts in Program Comprehension", in *Proceedings International Workshop on Program Comprehension (IWPC 2002)*, Paris, France, June 27 - 29 2002, pp. 271-278.
- [18] Rostkowycz, A., Rajlich, V., and Marcus, A., "A Case Study on the Long-Term Effects of Software Redocumentation", in *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Chicago, IL, September 11-17 2004, pp. 92-101.
- [19] Seidewitz, E., "What models mean", *IEEE software*, 20, 5, 2003, pp. 26-32.
- [20] Storey, M. A., "Theories, Methods and Tools in Program Comprehension : Past, Present and Future", in *Proceedings 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, MO., May 15 - 16, 2005 2005, pp. 181 - 191.
- [21] Storey, M. A., Bennett, C., Bull, R. I., and German, D. M., "Remixing Visualization to Support Collaboration in Software Maintenance", in *Proceedings Frontiers of Software Maintenance (FoSM 2008)*, Beijing, China, October 2008, pp. 139-148.
- [22] Teitelman, W., "' Do What I Mean': the programmer's assistant", *Computers and Automation*, 21, 4, 1972, pp. 8-11.
- [23] van Sickle, L. and Hartman, J., "Technical introduction to the first workshop on artificial intelligence and automated program understanding", in 1992, pp. 12-16.
- [24] von Mayrhauser, A. and Vans, A. M., "From Program Comprehension to Tool Requirements for an Industrial Environment", in *Proceedings IEEE Second Workshop on Program Comprehension*, Capri, Italy, July 8 - 9 1993, pp. 78 - 86.
- [25] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice*, 7, 1995, pp. 49-62.
- [26] Wittgenstein, L., *Philosophical Investigations*, New York, Mcmillan Publishing Co., 1953.
- [27] Xu, S., Rajlich, V., and Marcus, A., "An Empirical Study of Programmer Learning during Incremental Software Development", in *Proceedings 4th IEEE International Conference on Cognitive Informatics (ICCI 2005)*, Irvine, CA, August 8-10 2005, pp. 340-349.