

Program Comprehension as a Learning Process

Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

The paper describes the process of program comprehension from the point of view of constructivist theory of learning. According to this view, program comprehension starts with the pre-existing knowledge and continues through processes of assimilation and adaptation. The assimilation means that the facts encountered in the program are either added to the knowledge or rejected. Adaptation means that the existing knowledge is reorganized in order to absorb new facts. These processes are illustrated by a case study where the knowledge of the program is represented by UML class diagrams.

1. Introduction

Program comprehension is an essential part of software engineering, particularly software evolution and software maintenance. Software that is not comprehended cannot be changed, and hence it cannot evolve and cannot be maintained.

According to available data, program comprehension consumes most of a programmer's time [7], and hence the study of program comprehension is not only theoretically interesting, but it is also very practical. In order to improve program comprehension, whole research fields have been established, among them software documentation, visualization, program analysis, refactoring, reengineering, and so forth.

As an example of program comprehension, consider the well-known fact that some legacy systems contain business rules and other domain knowledge that is not recorded anywhere else. One such legacy system is documented in [8]. It is a system used by a major US insurance company to calculate premiums. The premiums are affected by state rules and slightly vary from state to state. The code of the system is the only place that records these rules. When programmers are

asked to evolve, maintain, or to re-implement such a program, they have to extract these rules first.

There has been at least quarter century of program comprehension research that produced many results. Prominent among them are the top-down and bottom-up theories of program comprehension.

According to the top-down theory [3], the programmers approach program comprehension through formulation of hypotheses. The hypotheses may form a hierarchy, as some hypotheses are refinements of other hypotheses. Programmers confirm or refute these hypotheses based on the facts found in the code. They retain the confirmed hypotheses and discard the refuted ones. They continue this process until they understand the whole program.

The opposite is the bottom-up theory of program comprehension [9], where programmers look for recognizable patterns in the code, called chunks. Smaller chunks may be nested within larger ones. The programmers recognize increasingly larger chunks, until the whole program is understood.

This paper presents a different view, where program comprehension is viewed as a learning process. In particular, the paper uses the constructivist theory of learning to explain program comprehension.

The learning that is the foundation of program comprehension is described in Section 2. Section 3 describes program comprehension from this perspective. Section 4 contains a related case study. Section 5 discusses other work, and Section 6 contains conclusions and future work.

2. Constructivist Learning

Constructivist theory of learning derives from the work of Piaget [12]. According to this theory, the learners actively and incrementally construct their knowledge. They always have some pre-existing knowledge and use it to deal with the new facts. The

two main processes of learning are called assimilation and adaptation. Assimilation means that the learners process the new facts and fit them into the existing knowledge. If the new facts fit imperfectly into the pre-existing knowledge, the learners reject or modify the facts that do not fit [12], [6]. Adaptation means that the learners reorganize their existing knowledge so that it can absorb the new facts.

Although most of [12] deals with children's learning, the constructivist theory of learning has been extended to adult learning and to the epistemology in general [6]. It is this latter aspect of constructivism that we use in this paper.

Conceptual maps [11] represent knowledge constructed by learners. They are graphs where nodes represent concepts and edges represent relationships between the concepts.

3. Program Comprehension

In the view of this paper, program comprehension is an instance of constructivist learning where the programmer learns new facts about the program. In the beginning, the programmer's knowledge of the program is incomplete but not empty, and the programmer learns new facts by reading the program. This reading is driven by the structure of programmer's pre-existing knowledge, so that the new facts can fit easily into the programmer's current comprehension. It cannot be driven by the structure of the code and start with the first line and end with the last, because fitting the new facts into the pre-existing knowledge in this way would be extremely difficult.

Theoretically, program comprehension should be captured in program documentation, but that is very rarely a case. In most of the cases, the programmer has to rely on other sources, like program manuals and the knowledge of colleagues on the project. These sources are usually sufficient for shallow program comprehension. However in the end, the only complete and reliable basis for the program comprehension is the source code itself.

The task of program comprehension is challenging when both the original creator and the current programmer come from the same background, but it is even more challenging when there is a cultural difference separating them [15]. In [15], it is documented how the programming culture changed over the years. The current programming culture is of course a part of the current programmer's pre-existing knowledge and the programmer understands the facts found in the code through this prism of the current

culture. This cultural difference plays an important role in the tasks of reengineering, where the old program has to be transformed into a new, while all functionality of the old program must be preserved.

As mentioned earlier, conceptual maps [11] are used to represent the concepts involved in learning. When today's programmers learn facts about the code, they use objects and classes. Class diagrams [2] became very popular for representation of object-oriented programs. In a class diagram, the nodes represent classes and the arcs represent relationships among the classes. The classes of the program are implementations of programming concepts and that makes them similar to conceptual maps. We believe that the class diagrams are suitable substitute for conceptual maps and can be used to represent the knowledge about programs.

In the following case study, an old Fortran program implemented in 1970's is comprehended in terms of object-oriented concepts, represented by UML class diagrams. The programmer's preexisting knowledge is the knowledge of the object-oriented programming, together with the knowledge of the program domain that was extracted from the user manuals. The programmers used the process of assimilation and adaptation, fitting the new facts gleaned from the code into the currently existing knowledge.

4. A Case Study

This section reports a case study of comprehension of an old Fortran program that illustrates the issues discussed in the previous section.

The program used in the case study is CONVERT program [15] that is a part of a larger FASTGEN geometric modelling system, used by United States Air Force. FASTGEN models targets such as vehicles and aircraft, and for that it uses primitive shapes like triangles, spheres, cylinders, donuts, boxes, wedges, and rods. CONVERT is the first program of the system and converts input model into a set of triangles, on which the remaining programs of FASTGEN operate.

CONVERT was implemented 1970's. Later it was ported to CDC 6600, CDC Cyber 176, CRAY Y-MP 8/2128, Digital Equipment Corporation VAX, and to personal computers. The current size of CONVERT is 2335 lines of Fortran77. Since the old program was written in 1970's, there was emphasis on overcoming limited resources of the computers and operating systems of that time.

After this long maintenance, the structure of CONVERT deteriorated to the point where the future

maintenance and evolution is very difficult. For that reason, re-implementation of the program in modern programming languages with modern program architecture became desirable. However, the old program contains valuable knowledge of the application domain that still has a great value for the user and must be extracted and preserved.

The team conducting the case study consisted of a graduate student and advisor [15]. Their pre-existing knowledge consisted of modern programming concepts like objects and classes. The facts that were gleaned from the program were assimilated into this pre-existing knowledge.

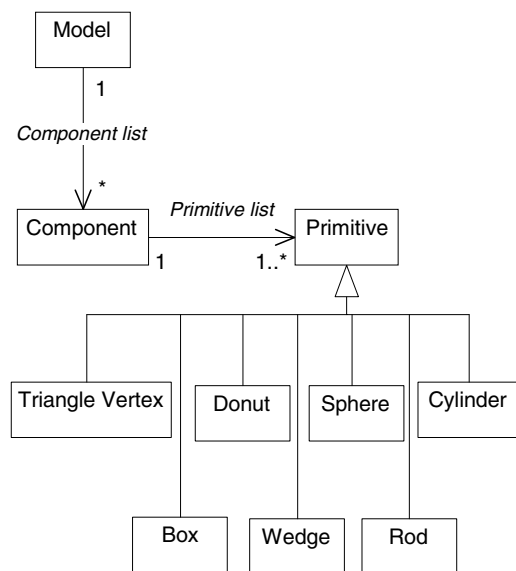


Figure 1. Pre-existing comprehension

Another part of the pre-existing knowledge was the knowledge of the program extracted from the user manuals [7]. At the beginning, the team reviewed the user manuals and created the preliminary model of program comprehension. This preliminary model had concepts of two different kinds: there were classes represented by the UML class diagram of Figure 1, and there was a list of 47 class methods. These two documents comprised initial understanding of the program and were a starting point for the process of program comprehension. The goal of the comprehension process was to understand the program in sufficient detail so that it could be re-implemented in object-oriented programming language C++. It was important to provide undiminished functionality to the user.

The pre-existing knowledge dictated the order in which the programmers searched for the new facts in the code. They located the old code of the concepts one by one, and then read them and understood them. For location, they used dynamic search, also called software reconnaissance [16].

Software reconnaissance, which is more completely described in [16], uses test cases as probes to locate program features. "Features" are selectable functionalities in the program, that is, functionalities that are invoked in response to the input data. They can be located in the code dynamically by two test cases for each feature, one "with" the feature and a similar test case "without" the feature. For example, to locate the "cylinder" feature in the program, the program would first be instrumented to trace the code executed on each test case. Then two test cases would be run, one with cylinder data and another without. The traces are compared to identify the "marker" code, which is executed only in the test with cylinder data. Programmers can find the feature's code by starting from the markers and tracing nearby data and control flow [16]. Software reconnaissance also discovers "common code", that is, the code executed by all test cases.

In the case study, total of 418 code parts (blocks or function entries) were instrumented. The initial tests covered 63.4% of the code, a fairly typical number for a functional test set. Of that, 13.4% was "common" code.

Once the common code or code of a feature was located, detailed reading followed. Study revealed that most of the common code of CONVERT reads records for the geometric model in 80 column format, typical for the punched cards of 70's. The remaining common code opens files, sets parameters, and performs other initializations.

The reading of the code of the features provided accurate understanding of the methods and classes. The team used process of assimilation [12] by adding new facts to the model. For example, the team discovered additional methods that were not mentioned in the user's manual, such as error checks. The team added these methods to the list and also added the corresponding test cases. Several additional classes were discovered as well, see Figure 2. The final products of this comprehension process included a UML class diagram, the test set, and accurate descriptions of the methods. These documents summarized the final program comprehension and were a starting point for program re-implementation.

In the process of assimilation, the facts in the old

code were either added to the knowledge, or rejected if they did not fit. There were some facts found in the code that were rejected as not fitting into the pre-existing knowledge. Those were the facts that were related to the now obsolete programming techniques of 1970's that dealt with the hardware and operating system limitations. Thus the team viewed the legacy code as consisting of two intertwined parts: a part that contains business value for today's user and is worth comprehending, and a part that was not worth the effort. The resulting model of Figure 2 contains only the part that represents business value for today's user.

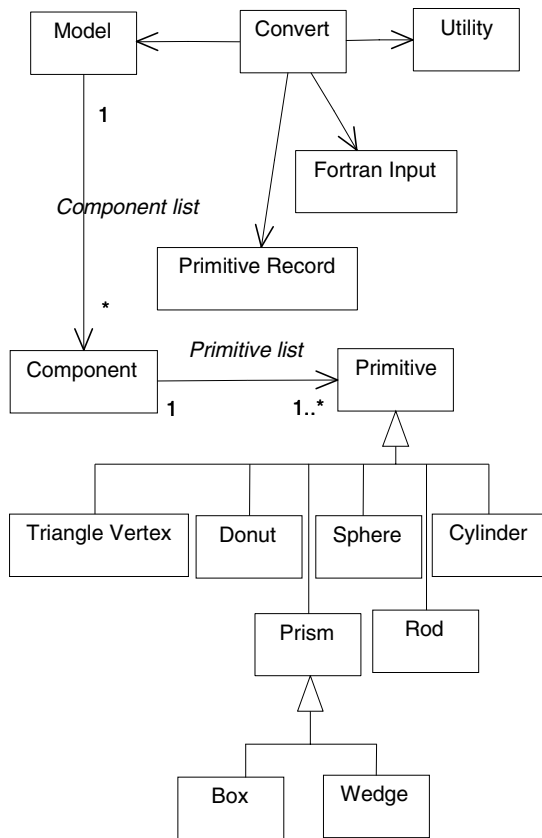


Figure 2. Final comprehension

The team rejected certain facts before actual comprehension effort, just based on the circumstantial evidence. For example, CONVERT contains code that is executed only when the arrays are almost full. This code most likely protects the geometric model data from array overflow; please remember that the geometric models have variable sized data while Fortran77 has only fixed size data arrays and therefore

this kind of protection was necessary. It is no longer necessary in modern programming languages with variable size data structures and therefore the related facts could be safely rejected. Another instance of a rejection was a large subroutine that was never executed in any test and was called only by invalid data. The team concluded that this is most likely an obsolete feature, not worth the effort to understand.

There was also process of adaptation [12] employed during the comprehension. The team noted that "Box" and "Wedge" share a substantial amount of code and decided to create a new class "Prism" that will contain this code, see Figure 2. The classes "Box" and "Wedge" now inherit from "Prism". This reorganization of the program knowledge was a direct result of the increased understanding of the program.

5. Related Work

Classic top-down [3] and bottom-up [9] theories of program comprehension are the reference point for most of the current program comprehension research. They were described in the introduction. A theory that combines both top-down and bottom-up appears in [10].

The case study of this paper extensively used the concept location technique of [16], where the programmer looks for specific programming concepts and their implementation in the code. The concept location especially fits with the constructivist theory of learning, because it allows the programmer to comprehend the facts in the order given by the structure of pre-existing knowledge, rather than the structure of the program.

The case study used concept location technique of [16]. In the literature, there are several additional concept location techniques. In [1], concepts are located by exploiting similarities between concept names and identifiers in the code. The technique uses textual pattern matching and the most popular tool used in practice is "grep" of Unix operating systems. Another technique is described in [4] where a search of static code is conducted by a programmer, who exploits data flows and control flows in the program. An overview of concept location techniques can be found in [13] and [14].

The similarities between program comprehension and learning were noted in [13] and [14] and this paper further develops this relation.

There are numerous publications on constructivist learning. They include classical work of Piaget [12]. A more recent and shorter summary appears in [6]. In

[11], structure of the knowledge constructed by the learner is modelled by conceptual graphs.

6. Conclusions and Future Work

The paper presented a synergy between the field of constructivist learning and program comprehension. It illustrated the approach by a case study where a specific program comprehension task is interpreted in constructivist terms.

Program comprehension can greatly benefit from this synergy. The field of constructivist learning accumulated many insights and theories that can be applied to program comprehension. At the same time, program comprehension provides a large test bed for constructivist learning theories. It is estimated that there are currently 70,000 active software projects in the USA alone. As mentioned earlier, more than half of this large effort is probably spent in program comprehension, and hence software industry is a large test bed for testing learning theories.

Future work includes research in knowledge representation for program comprehension. In this paper, we used an ad hoc representation consisting of class diagrams and a list of methods, but perhaps better representations can be found. This knowledge representation must support both assimilation and adaptation and record all facets of program comprehension.

This research direction promises both improvements in programming productivity, and at the same time it represents an interesting research challenges.

References

- [1] Biggerstaff, T.J., B.G. Mitbander, D.E. Webster, "Program Understanding and the Concept Assignment Problem", *Communications of ACM*, May, 1994, 72-78
- [2] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
- [3] Brooks, R., "Towards a Theory of the Cognitive Processes in Computer Programming", *Int. J. Man-Machine Studies*, Vol.9, 1977, 737-751
- [4] Chen, K., V. Rajlich, "Case Study of Feature Location Using Dependence Graph", *Proc. International Workshop on Program Comprehension*, IEEE Computer Society Press, 2000, 241-249
- [5] Fjeldstad, R.K., W.T. Hamlen, Application Program Maintenance Study: Report to Our Respondents, in G. Parikh, N. Zvegintzov, ed., *Tutorial on Software Maintenance*, 1982, 13-30
- [6] von Glasersfeld, E., *Radical Constructivism*, The Falmer Press, London, 1995
- [7] Jones, S.L., Aitken, E D., *Convert3.0 User's Manual*, ASI Systems International, Fort Walton Beach, FL: March 1994
- [8] Kozaczynski, W.,and Wilde, N., "On the Re-Engineering of Transaction Systems", *J. Software Maintenance*, Vol. 4, 1992, 143-162.
- [9] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension", *IEEE Software*, vol. 3, No. 3, May 1986, 41 - 49.
- [10] von Mayrhauser, A., A. Vans, "From Program Comprehension to Tool Requirements for an Industrial Environment", *Proc. 2nd Workshop on Program Comprehension*, July,1993, 78-86
- [11] Novak, J.D., *Learning, Creating, and Using Knowledge*, Lawrence Erlbaum Associates, Mahwah, NJ, 1998
- [12] Piaget, J. (1954) *The construction of reality in the child*. Basic Books, New York, 1959
- [13] Rajlich, V., "Program Comprehension and Domain Concepts", in T. Twoney, M. O'Brien, J.Donovan, ed., *Proc. Of the 1st INSERC Conference of Software Ergonomics*, ISBN 0-9541582-1-0, Limerick Institute of Technology Press, 2001, 65 – 73.
- [14] Rajlich, V., Wilde, N., "The Role of Concepts in Program Comprehension", to be published in *Proc. IEEE International Workshop on Program Comprehension*, 2002
- [15] Rajlich, V., N. Wilde, M. Buckellew, H. Page, Software Cultures and Evolution, *IEEE Computer*, September 2001, 24 – 29
- [16] Wilde, N., M.C. Scully, "Software Reconnaissance: Mapping Features to Code", *J. Software Maintenance*, 1995, 49-62