The Role of Concepts in Program Comprehension

Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI, USA
rajlich@cs.wayne.edu

Norman Wilde
Department of Computer Science
University of West Florida
Pensacola, FL 32514, USA
nwilde@uwf.edu

Abstract

The paper presents an overview of the role of concepts in program comprehension. It discusses concept location, in which the implementation of a specific concept is located in the code. This process is very common and precedes a large proportion of code changes. The paper also discusses the process of learning about the domain from the code, which is a prerequisite of code reengineering. The paper notes the similarities and overlaps between program comprehension and human learning.

1. Introduction

Program comprehension is an essential part of software evolution and software maintenance: software that is not comprehended cannot be changed. The fields of software documentation, visualization, program design, and so forth, are driven by the need for program comprehension. Program comprehension also provides motivation for program analysis, refactoring, reengineering, and other processes.

Because of its importance, program comprehension has been studied intensely, although many problems are still unresolved. Among the earliest results are the two classic theories of program comprehension, called top-down and bottom-up theories. The top-down theory explains program comprehension in the following way [5]:

> The programmer, when trying to comprehend a program, makes certain hypotheses and then confirms or rejects them based on evidence, the so-called beacon, found in the code. The confirmed hypotheses are retained, becoming part of the program's comprehension, while rejected hypotheses are discarded.

The bottom-up theory of program comprehension [17] is based on chunking. Chunks are parts of code that the programmer recognizes. A chunk has a specific meaning and often a name. Large chunks contain smaller chunks nested within them. The programmer pieces together his understanding of the program by combining chunks into increasingly large chunks. Both top-down and bottom-up program comprehension theories are complementary and have been combined into unified models [20].

In this paper, we want to present a different view of program comprehension, one that does not rely on the top-down vs. bottom-up dichotomy, but one that is based on the role of concepts. As programs have become larger, it has become ever less feasible to achieve complete comprehension. Instead, experienced programmers tend to use an as-needed strategy in which they attempt to understand only how certain specific concepts are reflected in the code [14]. They thus seek the minimum essential understanding for the particular software task at hand. Concepts play an important role in that, as we illustrate by several case studies summarized in the paper. It should be noted that concepts are also fundamental building blocks of human learning [21] and hence from our perspective, the disciplines of program comprehension and human learning have intriguing similarities and overlaps.

The role of concepts in software comprehension is described in Section 2. We illustrate the role of concepts by describing the process of concept location in Section 3 and related case studies in Section 4. Section 5 explores the problem of learning about the domain from the code and section 6 contains a related case study. Section 7 discusses some other approaches to the problem and Section 8 contains a summary and conclusions.

2. Concepts and their role

In most software engineering processes, complete comprehension of the whole program is unnecessary and often is impossible [16]. Change requests are often formulated in terms of domain concepts, for example "Add credit card payment to the point-of-sale system". The important task is then to understand where and how the relevant concepts are implemented in the code. Concept location is the starting point for the desired program change.

The concept location process assumes that the programmer understands the concepts of the program domain, but does not know where in the code they are located. For example, if we want to change an external viewer in a web browser, we have to find the location where the external viewers are implemented. For that we have to understand the concept of external viewer and how external viewers are used in a browser.

We should be aware that several simplified definitions of what is a concept appear in the literature. One is the popular idea that concepts are equivalent to objects in an object-oriented program. While it is true that in a well-structured object oriented program each class represents a concept (external viewer, credit card, and so on), the opposite is not true. There are many concepts of the program domain that are too trivial to have a class of their own. For example, the concept "payment" may be implemented as a single integer within class "sale" rather than having its own class.

Also, many concepts are spread across several classes. For example the "look-and-feel" of the application's user interface is implemented in several classes. Similarly, programmers increasingly use design patterns that typically involve collaborations of several related classes to implement a concept [9]. To locate such distributed concepts requires locating and marking all classes that participate. If the concept is going to change, all classes in this group may also change.

Another simplified notion of concept originates from the work of Birkhoff [3] and is very popular [19]. According to this definition, there is a fixed set of attributes and a concept is a specific subset of these attributes. The subsets constitute a lattice and therefore concepts also constitute a lattice. This notion again does not cover the full range of concepts encountered by the programmer, although in certain cases it can be very useful [19].

We should be aware that the notion of concept is often an involved one, see the discussion in [27]. In [21], p.36, a concept is defined as "perceived regularity in events or objects, or records of events or objects, designated by a label".

In our work, we use the following working definition:

> Concepts are units of human knowledge that can be processed by the human mind (short-term memory) in one instance.

Thus we would include in our definition both domain concepts that would be familiar to an end user ("credit card payment") as well as related high level design concepts ("iterator pattern used in the list of credit card holders"), and important error conditions that a user may be only dimly aware of ("network error while validating credit card").

Note that the set of concepts for a particular program is not fixed. The specification may use one set, additional concepts may be added in design, and some detailed concepts such as the error conditions may not emerge until programming. As well, one of the interesting aspects of software maintenance is the way new concepts can emerge as software is used in unexpected ways. Finally the lexicon used to describe concepts may vary as users, designers, programmers and maintainers use different words to describe essentially the same or similar concepts.

Concepts are an important part of human learning [21]. According to the constructivist theory of human learning [21], [22], humans actively construct their knowledge. They always have pre-existing knowledge that they extend based on new facts. Assimilation is a process where the new facts are incorporated without changing the pre-existing knowledge. Accommodation fits in new facts but requires reorganization of the pre-existing knowledge.

This theory is directly applicable to program comprehension. The programmers always have some pre-existing knowledge; otherwise the process of comprehension would not be possible. They assimilate new facts that easily fit into their pre-existing knowledge. When faced with facts that do not fit, they have to accommodate them. Programming knowledge has many components, but one of the most important ones is the domain concepts and their implementation in the code. The gaps in that knowledge are filled during program comprehension.

3. Concept location

Frequently in program comprehension the programmer understands domain concepts, but not the code. The knowledge of domain concepts is based on program use and therefore it is easier to acquire than knowledge of the code. For example when using a word processor, the user learns about cut-and-paste, fonts, and other concepts of the domain, but knows nothing about the implementation of these concepts in the program. Another source of knowledge of domain concepts is the user manual.

This original knowledge is the basis for further learning about concept implementation. All domain concepts should map onto one or more fragments of the code. The process of concept location is the process that finds this code.

Concept location is needed whenever a change is to be made. Change requests are most often formulated in terms of domain concepts. An example is "There is an error when trying to paste a text consisting only of capital letters, please correct." In order to make the required change, the user must find in the code the locations where concepts "paste" and "capital letters" are located - this is the start of the change.

Concept location has traditionally been an intuitive process greatly facilitated by the experience of the programmer. For example, most Software Engineering Masters students at the University of West Florida do a maintenance project, and we have observed how experienced students may locate the code to be changed in a few minutes, while others thrash for hours and do not seem to know how to begin. The experienced students may have some difficulty explaining exactly how they do what they do, since the answer to them is so obvious.

When intuition and experience fail to provide an immediate answer, programmers must become more systematic to locate the needed concepts. The most widely used technique is based on string pattern matching and uses the similarity of program identifiers to concept names [2]. So, for example, when searching for the location of cut-and-paste, the programmer may want to search for identifiers "cut", "cutPaste", "cut2", "xCut", "cutSelected", "cutText", and so on. When the appropriate identifier is found, the programmer studies the surrounding code to decide whether this is truly the location that implements the concept, or whether the similarity of names is just a coincidental correspondence. Also, the full extent of the concept's location must be established. The concept is implemented not only in the place where the identifier was found, but also in previous and following statements, the variables that are used in these statements, and so on.

A well-known example of a string pattern matching utility is "grep" available on most Unix systems; therefore this technique is sometimes called the "grep technique". In spite of its wide acceptance, it has serious deficiencies. It is based on the correspondence between the programmer's name for the concept and an identifier in the code. Both homonyms and synonyms create problems. The technique often fails, particularly when the concepts are hidden more implicitly in the code, or when the programmer is unable to guess the program identifiers.

Thus both the intuitive approach and the grep technique depend heavily on hints from the program's original developers. They must have used naming conventions that clearly encode domain concepts. They must have structured the program around these concepts and identified them in the code using meaningful and consistent names. As previously mentioned, a key problem is that, especially after many cycles of maintenance by diverse programmers, the vocabulary used to describe the software may no longer be the same as at its creation. Thus intuition and the grep technique are likely to become less useful on older and heavily maintained code.

These difficulties have motivated a search for alternative methods of locating concepts. One such approach is the *dynamic search* method, also called *software reconnaissance* [24]. It is based on the following idea:

> Some programming concepts are selectable, because their execution depends on a specific input sequence. For example cut-and-paste is selectable because the user of the word processor can select whether to use it or not. Selectable program concepts are called *features*. The code that implements features can often be found by executing the program twice: once with the feature and once without, and then marking the parts of the program that were executed the first time but not the second time. These parts are likely to be in or near code that implements the feature.

In order to find which parts were executed in which test cases, additional *instrumentation* statements that indicate which parts (functions, branches, or statements) were executed must be added to the program. Once the code has been instrumented, test cases have been run, and the appropriate code of the feature has been marked, the programmer again reads the relevant code in order to understand the program plans related to the feature. Deprez and Lakhotia have formalized and extended this method to show how adequate test data for a feature can be identified from a grammar of the program inputs [8].

Another technique of concept location is to search through the static code [6]. The search follows control flow and data flow dependencies among the program components. A typical scenario of the search goes top-down through control flow dependencies and is described in the following way:

> The functionality of the whole program is summarized in the top-most function `main()` or top class of the program. However this top class does not (and cannot) do everything, it delegates parts of its functionality to other classes. Hence if the top class does not implement the sought concept, it must be implemented by one of the classes or functions called by it. Since these called classes or functions are specialized, it is usually easy to decide which one does and which one does not contain the sought concept. Moving down through the call graph towards more and more specialized functions or classes, the programmer ultimately finds the classes or functions that participate in the concept.
>
> If the origin or destination of data is sought, then the programmer follows the data flows rather than control flows.

This static search is used when the results of the previous techniques either fail or have to be sharpened. When employing this technique, not only the functions and classes of the concept implementation must be understood, but also the functions or classes on the search path. However the understanding of these additional components does not have to be as accurate as the understanding of the components that participate in the concept implementation.

4. Case studies of concept location

We have performed a number of case studies of concept location to try to clarify the relationships between concepts and code comprehension. One of the more systematic studies applied the static search technique to the NCSA Mosaic 2.5 web browser [6]. The change request was to extend Mosaic to be able to handle a new type of audio files. The task is the following: Locate in the code where the type of the incoming file and its mapping to an external viewer are determined.

The task was decomposed into three subtasks. The first subtask was to find the function that opens a new window. The new window has the same browsing functionality as the old one; therefore the mappings must be copied immediately after the opening. We adopted the top-down strategy and started from the function main() and after several steps we located function mo_open_window() that opens a new window.

The second subtask was to find where and how the mappings are copied. It must be done sometime after the window opens and before any document is loaded. We continued the top-down strategy, starting in mo_open_window() and after several steps we found functions HTFormatInit() and HTFileInit() that copy the mappings.

Functions HTFormatInit() and HTFileInit() use several global variables. We needed to know where the values of these variables come from and this was the third subtask. Our strategy was to follow backward data flow to the source of these values. Ultimately we reached the variable HOME that is the location of the defaults; this is the location where the changes should be made.

The location process of this case study resulted in a partial comprehension of the system. Of the 984 functions in Mosaic, we visited only 22, about 2% of the code and that provided sufficient comprehension to be able to locate the concept and start the required change.

Other case studies used the dynamic search (software reconnaissance) technique described in the previous section [25], [10]. One such study was intended to provide an analysis of the domain concepts that appear in the program's user documentation, to clarify how user-level domain concepts map onto code.

The system studied was the analysis part of the ATAC test coverage monitor developed by Bellcore (now Telcordia) [11]. The program was approximately 10 K lines of code (raw line count) distributed into 24 C code files and 4 header files. The user documentation consisted of an extensive Unix-style "man" page, from which 24 different testable concepts were identified. A total of 77 test cases were written and used to mark code for each concept.

One interesting result shows how concepts are commonly *delocalized* in code [18]. Table 1 shows that 19 of the 24 concepts had code in two or more source files, indicating that the maintainer trying to understand the concept must integrate information from different and distant code fragments.

On the other hand, the study also showed how regularities in the design, and especially in the naming of functions and data, may greatly facilitate concept location. For example, ATAC is a test coverage tool for C programs, and so provides its users with information about how well a test set covers the functions, basic blocks, decisions and data flows (p-uses and c-uses[1]). Thus we have these five domain concepts (*function*, *block*, *decision*, *p-use*, *c-use*) in ATAC.

Table 1
Delocalization of Feature Code

Number of Files Containing Marked Code	Number of Concepts
0	3
1	2
2	8
3	5
4	2
5	0
6	0
7	0
8	2
9	2
Total	24

While each of these five concepts has code in several places in the program, ATAC had been designed so that there is one ATAC source file that concentrates on displaying information about each concept. The file names give important clues for concept localization. The file that contains most of the *function* concept was named `fdisp.c`, while the one with most of the *block* concept was `bdisp.c` and the one with most of the *c-use* concept was `cdisp.c`, and so on. The naming convention is obviously based on the first letter of the concept. Similar regularities were found in the names of functions within each of these files [25].

With this kind of parallel name structure, a maintainer can quickly learn how and where these different concepts are located. When one of the five concepts is understood, the others fall quickly into place. Presumably it is regularities such as these in well designed code that allow experienced software engineers to be successful with the intuitive approach described in section 3. Obviously however, such methods break down if the regularity is broken by a loosely coordinated design team or by ill-informed maintenance.

5. Learning about the domain from programs

In many software engineering situations, the programmer's knowledge of the domain is incomplete and the programmer may have to learn more about the domain from the program.

This process is rarer than the process of concept location, but it is still important, especially during reengineering. It is well known that some legacy programs contain business rules and other domain information that may not be available anywhere else. There may be algorithms or formulas that solve problems successfully, but these algorithms or formulas are not recorded anywhere other than in the code.

For example Kozaczynski and Wilde mention a legacy system used by a major US insurance company [15]. The rules for calculating insurance premiums are subject to differing state laws and many slight variations had accumulated over the years. Most of these variations are undocumented and can only be discovered through analysis of the system code.

When programmers are asked to reengineer or even to re-implement such a program from scratch, they still need all the knowledge that is contained in the old program. In order to recover that knowledge, the programmers have to rebuild the hierarchy of domain concepts and the details of their implementation based on the old program. One methodology for this is described in [26].

The methodology starts with a study of user manuals or similar documents and based on them, creates a first approximate version of the domain model. After that,

[1] p-uses and c-uses are different kinds of data flows used in data flow testing [11]. A p-use is a use of a variable in a predicate, such as an if statement. A c-use is a use of a variable in a computation, such as an assignment statement.

the concepts of the domain are located in the code one-by-one and their details are studied. From that a more accurate comprehension of the domain emerges. That comprehension is then used in reengineering.

6. Case study of learning about the domain

An example of learning about the domain from the code is a case study of the reengineering of the CONVERT program [26]. CONVERT is part of the FASTGEN geometric modeling system that models solid objects using primitives like triangles, spheres, cylinders, donuts, boxes, wedges, and rods. The United States Air Force uses it to model the impact of explosions and projectiles on targets such as vehicles, aircraft, etc. CONVERT transforms models into triangles, as required by other FASTGEN programs [13]. It is written in Fortran77 and consists of 2335 lines.

CONVERT has a long maintenance history, going back to the original program of 1978. Since that time, it has been ported to several hardware platforms, including CRAY Y-MP 8/2128 and Digital Equipment Corporation VAX. Lately, CONVERT was ported to personal computers.

Reengineering of the code has become desirable because after such prolonged maintenance, the structure of CONVERT is obsolete and very hard to maintain. It has poor modularity, with large, non-cohesive subroutines. Most of the data is held in large COMMON blocks, each referenced by many subroutines. The control flow is tangled, with large unstructured loops created by backward-branching GOTO statements. There are obsolete program plans that were necessary in early operating systems. For example there is batching of input/output into blocks of 200 records for greater efficiency, the use of scratch files to avoid overflowing of fixed size arrays, packing of multiple control flags into a single integer to save memory, and so on. Nevertheless the program contains valuable knowledge of the domain that still has a great value for the user. This knowledge must be preserved during reengineering.

As the first step, we reviewed the CONVERT user's manual [13] and created a tentative domain class model. We also extracted an initial list of 47 CONVERT features. These two documents represent initial understanding of the program domain.

The next step was location of the features in the code. For that we used dynamic search (software reconnaissance). Two test cases were identified for each feature, one "with" the feature and a similar test case "without" the feature.

A 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, executed on every invocation.

The study revealed that most of the common code of CONVERT reads 80 column records for a geometric model. The remaining common code performs initializations, such as opening files, setting parameters, etc.

After analyzing the common code, we located and learned details of the individual features, one at a time. As features were understood, they were assigned to classes in the domain class model, adjusting the model as necessary to accommodate our increased understanding of the application. For example, we discovered some additional features that either were not mentioned in the user’s manual or were missed during the reading, such as error checks. These features were added to the list and we added the corresponding test cases. Final products of this process included a UML class diagram, the test set, and descriptions of the features. These documents are a starting point for program reengineering.

7. Other Work

There are two other threads of research and Software Engineering practice that should be mentioned in the context of concept location.

The first of these is *change impact analysis*. Impact analysis is a long established field in software maintenance, which attempts to identify the impacts or "ripple effects" that a change in one part of a program may have on another. It is not possible here to give a complete survey of this field. The reader is referred to the excellent set of papers and the bibliography in [4].

While most impact analysis has traditionally focused on code, some approaches take into account the whole range of documents that form part of a software system, including specifications and design. In this case, impact analysis blends into the concept of *traceability*, the ability to trace specification to design and design to code. If an impact analysis tool provides traceability information then it may be possible to trace specification concepts to the relevant code.

While traceability is undoubtedly useful, we should note two limitations from the point of view of concept location. The first is that maintaining traceability information over a system's life cycle usually requires considerable manual work, and thus is often slighted under the time pressures often associated with software development and maintenance. Second, the traceability information will almost certainly be expressed using the set of concepts perceived at specification time. As noted in section 2, the concept set and the lexicon used to describe it often change substantially over the life of a system.

The second thread related to concept location is that of *fault location*. Researchers working on improved debugging techniques have evolved techniques for locating software faults by comparing execution slices of different test cases. These techniques are often quite similar to the dynamic search method described in section 3. Perhaps a fault could be considered to be an unwanted concept?

Again it is not possible to completely survey the fault location literature here. The earliest paper on this topic seems to have been by Collofello and Cousin, who instrumented decision-to-decision paths in order to locate faults seeded into Pascal programs [7]. Other more recent work includes [1] and [12].

The relationship between human learning and program comprehension was noted in [23].

8. Conclusions

In this paper, we explored domain concepts and their role in program comprehension. There are several open problems in this research.

The location techniques mentioned in this paper need additional refinement. The static technique is based on program analysis. There are many open problems in static program analysis, starting with algorithms of pointer aliasing, discovery of hidden dependencies [28], and so on. The software reconnaissance method relies on instrumentation and recompilation of the program, followed by execution of the instrumented version with feature-tagged data. These steps may be awkward under the time pressures of a commercial production environment; smoother tool support would be very useful.

Also of interest would be an integrated software tool that would combine capabilities of several techniques of concept location: pattern matching, static search, and dynamic search, and allow the programmer to use the most appropriate one for the specific situation.

Another interesting set of problems arises from the similarity between process of constructivist human learning and program comprehension. In the theory of human learning, conceptual maps are used to describe human knowledge [21]. We speculate that these approaches can be applied to program comprehension and may offer additional interesting insights and techniques.

An attractive aspect of this field is the fact that research on the identification and location of domain concepts in software promises both improvements in programming productivity, and at the same time it provides significant research challenges.

References

[1] Agrawal, H., Horgan, J., London, S., Wong, W., "Fault Localization using Execution Slices and Dataflow Tests", *Proc. Sixth International Symposium on Software Reliability Engineering*, IEEE Computer Society, Oct. 1995, pp. 143 - 151

[2] Biggerstaff, T.J., B.G. Mitbander, D.E. Webster, "Program Understanding and the Concept Assignment Problem", *Comunications of ACM*, May,1994, 72-78

[3] G. Birkhoff, *Lattice Theory*, American Mathematical Society, Providence, RI, 1940

[4] Bohner, S and Arnold, R, *Software Change Impact Analysis*, IEEE Computer Society, Los Alamitos, CA., 1996.

[5] Brooks, R., "Towards a Theory of the Cognitive Processes in Computer Programming", *Int. J. Man-Machine Studies*, Vol.9, 1977, pp.737-751

[6] Chen, K., V. Rajlich, "Case Study of Feature Location Using Dependence Graph", *Proc. International Workshop on Program Comprehension*, IEEE Computer Society Press, 2000, pp. 241-249

[7] Collofello, J., and Cousin, L., "Towards automatic software fault location through decision-to-decision path analysis", *Proceedings National Computer Conference*, 1987, pp. 539 - 544.

[8] Deprez, J. and Lakhotia, A., "A formalism to automate mapping from program features to code", *Proceedings 8th International Workshop on Program Comprehension - IWPC 2000*, IEEE Computer Society, Los Alamitos, CA., June 2000, pp. 69 - 78.

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[10] Gunderson, A., Wilde, N. and Casey, C., *Locating Features in Interbase: A Software Reconnaissance Case Study at GTE Government Systems*, report SERC-TR-77-F, Software Engineering Research Center, University of Florida, Gainesville, FL 32611, March, 1995.

[11] Horgan, J., London, S., Lyu, M., "Achieving Software Quality with Testing Coverage Measures", *IEEE Computer*, Vol. 27, No. 9, September 1994, pp. 60-69.

[12] Jones, J, Harrold, M., Stasko, J., "Visualization for Fault Localization", *Proceedings of the Workshop on Software Visualization*, May 2001, http://www.cs.brown.edu/research/softvis/Contents.htm (URL current January, 2002).

[13] Jones, S.L., Aitken, E D., *Convert3.0 User's Manual*, ASI Systems International, Fort Walton Beach, FL: March 1994

[14] Koenemann, J., and Robertson, S., "Expert Problem Solving Strategies for Program Comprehension," *Proceedings of the Conference on Human Factors in Computing Systems, CHI'91*, ACM Press, pp.125-130, May 1991.

[15] Kozaczynski, W.,and Wilde, N., "On the Re-Engineering of Transaction Systems", *J. Software Maintenance*, Vol. 4, 1992, pp. 143-162.

[16] Lakhotia, A., "Understanding Someone Else's Code: An Analysis of Experience", *J. Systems and Software*, 1993, pp. 269-275

[17] Letovsky, S., "Cognitive Processes in Program Comprehension", *Empirical Studies of Programmers*, Eds. E. Soloway and S.S. Iyengar, 1986, pp.58-79

[18] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension", *IEEE Software*, vol. 3, No. 3, May 1986, pp. 41 - 49.

[19] Lindig, C., G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Theory", *Proceedings of International Conference on Software Engineering*, IEEE Computer Society Press, 1997, 349-359

[20] von Mayrhauser, A., A. Vans, "From Program Comprehension to Tool Requirements for an Industrial Environment", *Proc. 2nd Workshop on Program Comprehension*, July,1993, pp.78-86

[21] Novak, J.D., *Learning, Creating, and Using Knowledge*, Lawrence Erlbaum Associates, Mahwah, NJ, 1998

[21] Piaget, J. (1954) *The construction of reality in the child*. New York, Basic Books.

[23] Rajlich, V., "Program Comprehension and Domain Concepts," in T. Twoney, M. OBrien, 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.

[24] Wilde, N., M.C. Scully, "Software Reconnaissance: Mapping Features to Code", *J. Software Maintenance*, 1995, 49-62

[25] Wilde, N. and Casey, C., *Case Studies in Software Reconnaissance*, report SERC-TR-78-F, Software Engineering Research Center, University of Florida, Gainesville, FL 32611, May, 1995.

[26] Wilde, N., Buckellew, M., Rajlich, V., "A Dynamic Analysis Methodology for Reengineering Fortran to C++", to be published

[27] Wittgenstein, L., *Philosophical Investigations*, Mcmillan Publishing Co., New York, 1953

[28] Yu, Z., Rajlich, V., "Hidden Dependencies in Program Comprehension and Change Propagation", *Proc. International Workshop on Program Comprehension*, IEEE Computer Society Press, 2001, 293-299