Expressiveness and Effectiveness of Program Comprehension: Thoughts on Future Research Directions

Jonathan I. Maletic*, Huzefa Kagdi**

*Department of Computer Science
Kent State University
Kent Ohio 44242

**Department of Computer Science
Missouri University of Science and Technology
Rolla Missouri 65409

{jmaletic, hkagdi}@cs.kent.edu

Abstract

A number of research challenges in the area of program comprehension are presented. The expressiveness and effectiveness of program comprehension are discussed, and research directions are organized along these two axes. Both fundament research issues are raised along with new applications for program comprehension methods. The work advocates the investigation of better measures, further empirical studies, and controlled experiments to assess the effectiveness of program comprehension techniques

1. Introduction

Program comprehension is a major activity during software maintenance and evolution and is driven by the need to change software. Likewise, software change is driven by the need for new functionality, the correction of errors, or changes to libraries and platforms. Here we are particularly interested in the tools, techniques, and theories that support the program comprehension process.

There has been a large amount of research directed at the problem of program comprehension during the past 20 years. A relatively recent survey of the literature and research endeavors was presented by Storey in a keynote address at the International Workshop on Program Comprehension (IWPC) in 2005. The accompanying survey paper [20, 21] provides an excellent overview of the field. Her work gives a nice introduction of the related cognitive theories and describes what issues have been addressed and what issues are still open. There is also 20 years of research described in the workshop/conference on program comprehension i.e., the International Conference on Program Comprehension (ICPC)[1]. Other software-engineering venues such as the International Conference on Software Maintenance (ICSM), the International Conference on Automated Software Engineering (ASE), the International Conference on Software Engineering (ICSE), and the Working Conference on Reverse Engineering (WCRE) also regularly publish papers in the area of program comprehension.

Instead of making yet another attempt to recapitulate a large body of literature this paper focuses on open issues and future research directions in the area of program comprehension. In particular, we focus on the *expressiveness* and *effectiveness* of program comprehension research, and identify specific areas that require further investigation in the quest for techniques to support developers during maintenance and evolution. Expressiveness is discussed in terms of what we are trying to comprehend and effectiveness is concerned with the usefulness of the method. As we will see, there are open research problems along both axes.

Additionally, we focus on research that supports the comprehension of large, real-world, software systems by expert programmers. While understanding how novices comprehend or how small programs are comprehended is an important issue for education and basic understanding of cognition we leave it out of this discussion.

1.1. Expressiveness

In program comprehension we must be specific about why we are trying to comprehend (i.e., task), what we are trying to comprehend (i.e., object), and who's trying to comprehend (i.e., subject). The object of comprehension may be as small as a single function or as large as the entire software system. Examination of the source code alone maybe sufficient in function-level understandings, whereas understanding the entire system may span across multiple types of artifacts including source code, design documentation, and requirements. Typically, the object of comprehension is dependent on the task of interest. That is, there must be some reason to drive the need to comprehend an artifact. For example, are we trying to localize a bug/defect or assess a potential or existing change to an API? Most often we are looking for a concept or specific feature in the software that is related

[1] See http://www.program-comprehension.org

to some change that must take place. Outside of educational purposes, other reasons for comprehension maybe in support of a code inspection or general familiarization with the software.

Here we take the position that software change is best accomplished via an incremental process and as such most program comprehension activities are also incremental in nature. This begs the question as to whether program comprehension methods should be aimed at complete understanding or as needed understanding. Researchers are currently pursing both avenues but the difference has not been highlighted. This is a basic research question about expressiveness, which is still open.

Along with this fundamental research question goes more application oriented research avenues. Here we identify three specific topics. In particular we feel the area of understanding libraries, particularly generic libraries, is currently not being addressed. Additionally, the role of formal methods in support of program comprehensions is still an unaddressed question. Lastly, a more synergistic approach to language design in support of program comprehension is needed.

These are addressed in more detail in a following section but now we elaborate more on effectiveness.

1.2. Effectiveness

The effectiveness of program comprehension tools mainly deals with their quality (the litmus test) and eventually developer acceptance (the ultimate test). That is, will the newer tools prove to be good enough in the minds of developers to replace the incumbent tools (e.g., ubiquitous utilities such as *grep* and *diff*) in their development environment? Will developers accept a tool based on the (new) technology? Answers to these two questions will tell quite a lot about the effectiveness of a given program comprehension method. There is a need for better measures to assess program comprehension methods and tools. We need to develop quantitative measures and conduct empirical studies to better support effectiveness. We describe some recent work on this subject and highlight missing components.

The paper is organized as follows. First, we discuss research avenues for improving expressiveness in Section 2. Following that in Section 3 we do the same for effectiveness and wrap up with conclusions.

2. Improving Expressiveness

With the adoption of evolutionary/incremental development models, such as agile approaches [1], comes the need for as needed comprehension tasks, such as concept or feature location. The premise here is that the developer need to understand only a small part of the software in order to address a given change request.

At the other end of the spectrum are systematic and complete models for comprehension. The work in reverse engineering or inferring an ontology of a given software system or group of related systems is a prime example of a purely systematic approach.

This broad differentiation of comprehension approaches can be traced back to the work by Littman [11]. However, today we work in the context of very large software systems under continuous change. The scope of what we are trying to comprehend must now be more directly tied to the types of tasks being undertaken. In the case of maintenance tasks such as adding a new feature, fixing an error, or making a adaptive change in response to an API, hardware, or compiler change, we need tools that support as needed comprehension. In the case of complete design recovery or reverse engineering we need a more systematic approach. The later has been studied in a much more in depth manner, while the former is still in need of investigation. We feel this is an important aspect to improve the effectiveness of program comprehension methods.

Along with the basic research on expressiveness we now discuss some specific application areas for further researcher efforts in program comprehension. The first deals with the issue of how we comprehend generic libraries. Following that is a discussion on formal methods and program comprehension.

2.1. Comprehension & Generic Libraries

The fields of software engineering and programming languages are intertwined in interesting and obvious ways, but interaction between the two communities are not as synergistic as one would like. For the most part, programming language design is done without much thought to software engineering issues. This is particularly apparent in C/C++ with its use of the preprocessor. The preprocessor is widely used to solve many practical software engineering issues such as portability but lies outside the actual language.

On the forefront of the SE/PL divide are software libraries. Libraries have become the mainstay of not only large-scale software design and development, but also of programming language. Java would be pretty much just a (slower) variant of C++ without its very comprehensive set of libraries. Libraries such as the C++ Standard Template Library (STL) are now widely viewed as part of C++ and necessary for development (unless an alternative such as BOOST[2] is at hand).

Unfortunately, anyone who has used a library finds them difficult to learn, use, and understand. Our position here is that this is not a fault of the language or library designers, but a fault of the program comprehension

[2] See http://www.boost.org

community in not properly studying the issue of library design for comprehendability.

For instance, the upcoming version of C++ (currently being called C++0x) will introduce a broad array of new features ranging from minor syntactic cleanups to support for additional programming paradigms. One such feature aims to augment and extend current support for generic programming, a paradigm well suited to the development of abstract datatypes and generic algorithms. In the current version of C++, generic programming is predicated upon language features such as templates, overloading, argument-dependent lookup, and various programming techniques such as template metaprogramming. The new features for generic programming, in C++0x[3], are centered on the introduction of the *concept*, a first-class linguistic entity for expressing requirements on sets of types. Concepts can be used to constrain the sets of types over which generic algorithms (i.e., function templates in C++) are instantiated.

With broader support for generic programming in C++, we expect it to become even more widely used to create generic libraries for any number of domains. It will also improve the quality and usability of generic libraries. However, the development and maintenance of generic libraries is not without its difficulties. With respect to generic algorithms, the correct identification of requirements (concepts) on the types that the algorithm operates on is critical. Such requirements dictate what sets of types can or cannot be used with the generic algorithm. For example, one can use pointers in the place of most iterators, but not integers since they cannot be dereferenced. This implies a significant effort is on the horizon to convert existing C++ generic libraries to use concepts.

We feel that the process of migrating and reengineering existing C++ libraries to C++0x will greatly benefit from automated concept identification. We have developed an approach [22] to automatically identify concepts within function templates. Our approach, analogous to type inference, identifies the possible data abstractions represented by the template parameters of a function template. Analyzing the operators, functions, and types used on or by template parameters yields a set of concepts that describe their fundamental abstractions. Additional analysis is used to further refine this set of concepts and finally determine the set of concept instances that best represent the use of each template parameter within the function.

We prototyped the approach and ran it against a number of STL algorithms and compared the results against the requirements proposed for these algorithms in C++0x's STL description. We also ran the prototype against small variations of the same algorithm to determine the impact on the requirements based on differences in implementation. These evaluations show that the approach is not only a viable tool for assisting in the reengineering of existing generic libraries, but also provides useful insight into the construction and evolution of concept hierarchies and the design of new generic libraries.

This type of tool support for aiding in the comprehension of existing generic libraries is likely to be useful for tasks other than migration. Tools to help validate and understand concept hierarchies will also assist developers in the construction of these hierarchies as well as the users of them.

2.2. Formal Methods & Comprehension

Formal methods have played an important role in the development of mission critical software that requires an extreme level of quality assurance. The advent of model checking has made the use and application of formal methods more widespread and practical to a degree. Douglas Smith's keynote [16] address at IWPC 2005 highlighted that the use of formal derivation implies comprehension. Smith advocates that formal approaches centered on requirements specification with automated tools to generate code via design refinements have a side effect of resulting in rationale to understand how the system was evolved. As the system evolves through this process the history of design decisions is saved.

Formal methods allow for an alternative representation of a program. These alternative representations, by definition, assist in comprehending the program. We've used alternative visual representation of program in a similar manner i.e., to assist in our understanding of a software system.

Contrary to popular belief, formal methods, abet light--weight, have been seeping into common everyday usage by developers and in the classroom. The use of pre and post conditions, invariants, class invariants, and assertions are a very common mechanism for documenting source code. This idiom can be used in a very formal first order predicate calculus sort of manner to very informal natural language statements. The benefit is in the concise re-expression of the source code with the side effect of supporting comprehension (not to mention verification and/or validation).

A notable synergistic melding of software engineering and programming languages has resulted in built-in support and automated checking of pre/post conditions, invariants, and assertions (of course in limited ways). Spec#[4] and JML[5] are two such examples.

[3] See http://www.open-std.org/JTC1/SC22/WG21/ for the complete concepts proposal for C++0x

[4] See http://research.microsoft.com/SpecSharp/

[5] See http://www.jmlspecs.org

Spec# is an extension of C# and being developed at Microsoft Research. Spec# adds specification constructs to the language such as pre and post conditions, non-null types, and some high level data abstractions. These specifications are then checked statically via a program verifier (called Boogie). Moreover, the development environment directly supports the intent of the language by displaying unsatisfied specification with MSWord like spelling/grammar error underlining. An example of the Spec# pre-condition construct *requires* is below:

```
class ArrayList {
  public virtual void
       Insert(int index, object value)
  requires 0 <= index && index <= Count;
  requires !IsReadOnly && !IsFixedSize;
}
```

JML is a formal specification language for Java. Specifications such as pre and post conductions (with constructs *requires* and *ensures*) are written in a form similar to Java annotation comments. JML toolset include JML compiler, runtime assertion checker, formal verification via theorem proving, and static checkers. Formal modeling languages such as Spec# and JML allow for comprehension at an abstract, conceptual level of source code without the need to delve into the specific implementation details. This may also serve as an additional, intermediate layer that may aid in improved comprehension of source code before having to resort to the next high-level abstractions such as design documentation; thus potentially filling the comprehension gap between design and code.

We now examine the effectiveness aspects of program comprehension.

3. Improving Effectiveness

Empirical studies designed as surveys, case studies, and experiments are a rigorous means of evaluating and comparing software engineering tools that support program comprehension via interactive search, re-documentation, design recovery, visualization techniques. Studies designed to validate program comprehension techniques typically involve human subjects answering (performing) a set of questions (tasks). Arguably, irrespective of the very careful soundness consideration in the study design (e.g., selecting appropriate tasks and their distribution among participants), the validity of the achieved results directly corresponds to the type and quality of the used measures.

Traditionally, objective measures such as the accuracy/level of the response and time needed are collected from these studies [13, 14]. Additionally, subjective data such as a subject's experience, comments, preferences, and any other feedback are also collected. These measures are used directly or indirectly to draw conclusions and/or meet other objectives of the performed study. A wide majority of these traditional measures are collected retrospectively. For example, human subjects are asked to report their final answers on the completion of a given task and their response time is recorded. We term such measures as *black box* measures as they only record the final outcome after a specific task conclusion. That is, no other data is collected, at least not implicitly, while a human subject is performing a given task. There is almost no measurement taken that helps understand how and why a subject chose a particular (correct or incorrect) answer or solution. Additionally, black box measures raises a potential threat to the validity of the study, namely the match/disparity between the subjects' responses on completion of a task and the "reality" they observed while performing that task. For example, a subject may forget to report (or misreport) an observation after a lengthy task. Alternatively, subjects could be asked to note their observations while working towards their answers, albeit at the potential risk of obtrusiveness and distraction.

An alternative means of measurement is to use technology such as eye-tracking equipment to implicitly collect a subject's activity data in a non-obtrusive way while they are performing a given task. The equipment collects three forms of pertinent data including the eye gazes on the visual display and an audio/video recording of the subject's session. Eye gazes are substantiated with the measurement of various eye movements. This eye movement data could provide a valuable insight as to how and why subjects arrive at a certain solution. Therefore, we term the eye gaze measures collected from eye tracking as *white box* measures. These measures add a new additional dimension in assessing visualization tools claim of supporting software comprehension tasks.

To exemplify both black box and white box measures we briefly discuss two fairly recent studies. Both use human subject with the first being a systematic experimental study done by Cox that uses questionnaires to assess any difference in how males and females comprehend and solve programming problems. The second is a study we conducted using eye tracking equipment to assess the comprehendability of a different UML class diagram layouts.

3.1. Measuring Comprehension Differences

A number of studies [2, 3, 5, 8-12, 15, 17-19, 24-28] that involved human subjects were done in the late 1980's and early 1990's that form the basis of what we term as a mental model of program comprehension. However, much of this work used very small programs and studied novice users. Little work has been done in the study of experts attempting to understand real large programs. As such there is still a need for continued investigation through controlled experiments and empirical studies of how humans comprehend software.

However, this work is quite difficult to undertake. To properly run controlled experiments with human subjects a researcher (or team) needs expertise in both computer science and cognitive psychology. One particularly interesting recent study [4] addresses the issue of how the sexes differ in how they comprehend programs. This is a particularly pertinent issue with the United States National Science Foundation (NSF) due to the fact that very few women, in the U.S.A., pursue degree programs in computer science and computing in general[6]. The general consensus is that the lack of women in technology is a looming crisis. The Fisher, Cox, Zhao study [4] gives some evidence to support this belief.

Their study found evidence that females tend to use bottom-up program comprehension strategies and males tend to use top-down strategies. This also corresponds to observed difference in male and female spatial cognition. This is to say that males and females both solved comprehension tasks equal well but tended to use different strategies in the problem solving process.

The implication of this study is that a software development team should have a mix of males and females. Some problems may be solved easier with a top-down or bottom-up approach and lacking individuals, either male or female, without these tendencies could impede progress towards a solution.

This study is notable not only on its broader impacts but also on the experimental design and attention to science. It is a recent example of a model for constructing and conducting a control experiment investigating how we understand programs and gives us a better understanding of the effectiveness of program comprehension strategies.

3.2. Eye tracking as a Measure

Here, we briefly discuss the concepts of eye tracking in the context of our recently reported study on assessing UML class diagram layouts [29]. Also, we discuss how eye movement measures could be used to assess exploration, examination, and navigation support.

The underlying basis of an eye tracking equipment is to capture various types of eye movements that occur while humans physically gaze at an object of interest. Fixation and saccade are the two most common types of eye movements.

Definition: *Fixation* is the stabilization of eyes on an object of interest for a certain period of time.

Definition: *Saccades* are quick movements that move the eyes from one location to the next (i.e., refixates).

[6] See http://cra.org for discussions on the dichotomy of women in computing fields.

Definition: *Scanpath* is a directed path formed by saccades between fixations.

The general consensus in the eye tracking research community is that the processing of visualized information occurs during fixations, whereas, no such processing occurs during saccades [7]. Humans use saccades to locate interesting parts in a visual scene to form a mental model.

Figure 1 shows the recording of eye positions superimposed on a UML class diagram. The numbered circles represent fixations and lines between them represent saccades. The size of a fixation (i.e., area of a circle) is proportional to its time duration. The numbering of circles represents the ordering of fixations. For example, in Figure 1, the fixation labeled with the number 35 on the class *NTuple* happened before the fixation labeled 36 on the class *NTupleController*. That is, the class *NTuple* was looked at before the class *NTupleController*. The scanpath in this case is directed to the left and downwards. A big circle on the class *PyNTuple* shows that a large amount of time was spent on this class. The eye-tracker captures fixation and saccades in the form of XY coordinates of the visual screen from which we can determine what was being looked at in a visual presentation.

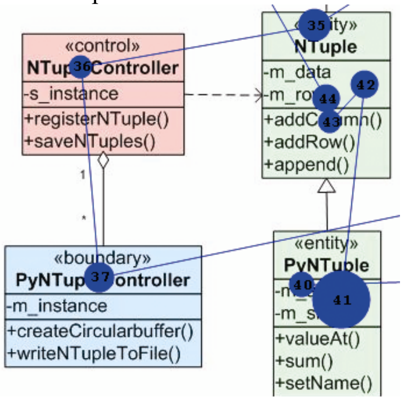

Figure 1. Gaze Information on a UML Class Diagram. Fixations are represented with circles and saccades with lines.

We used a *Tobii 1750* eye-tracker[7] to capture eye movements and collect eye gaze data. In this equipment, the two cameras used to track the eyes are built into a 17 inch flat-panel screen. Therefore, no restraints such as wearing a headband or goggles are placed on the human subject. This was not the case in older eye tracking equipment. This provides a normal computer-operating environment during the study. Moreover, the *Tobii 1750* eye-tracker is very accurate with an error rate of less than 0.5 degrees and a sampling rate of 50MHZ. Software that records the XY screen coordinates of eye gazes and supports analysis of eye movements is also provided

[7] See http://www.tobii.se

along with the eye-tracker system. An audio/video recording is also made of each study session.

Now, we describe the use of white box measures such as fixations and saccades in the evaluation of UML class diagram layouts. From layout perspective, the support for exploration, explanation, and navigation are of general interest.

An exploration activity deals with how subjects perform searches on the UML class diagram to locate objects required for a given task. The number and size of fixations could help identify areas of the layout that smoothly assisted or created bottlenecks in the exploration activity. Also, the scanpaths provide the order and directionality information in which the search space was traversed. For example, do the recorded scanpaths justify a particular layout's strategy of placing certain classes at a particular position? Were only the relevant classes immediately visited and only once?

An examination activity deals with how subjects visualize, in detail, whole or parts of a specific class and relationship. In our experience, fixations can be recorded at the granularity of a specific line (i.e., class, attribute, method names). Thus, fixations could be used to assess questions/tasks that are related to a specific class. Also, the durations of fixations give information about which parts of a specific class receive the most attention.

A navigation activity deals with how subjects move from one object of interest to the next after their discovery. Once again fixations and saccades could be used to justify a layout's strategy in supporting navigation.

In our previous study [29], we used the number of fixations as an indicator of the required human effort. Fewer number of fixations on a layout means that the subject needs less effort to answer the associated question. If the total number of fixations is high then the classes and relationships are possibly laid out in a way that leads to an inefficient visual exploration, explanation, and navigation. Such poor arrangement spans the attention of a subject across a number of objects instead of systematically narrowing down to only the relevant area of interest.

In another instance, Uwano et al. [23] used eye tracking to study the subjects performing source code reviews. Fixations were used to analysis how and which lines of source code were being examined. They observed that reviewers typically "scanned" the entire code before narrowing down their detailed focus and efforts on particular parts (lines) of interest. Also, reviewers who did not spend much time towards the initial scan of the code took longer than those who did to find the defects (intentionally planted in the code for the study).

The basic premise of eye tracking methods lays in the strong eye-mind hypothesis [6], which states that human gazes directly correspond to their thinking and cognitive process. We believe that white box measures from eye tracking are a promising step towards developing objective metrics for software comprehension and cognitive load. The measures should be used synergistically with the traditional black box measures in the empirical assessments of software Visualization (generally engineering) tools. Considering a number of recent advancements in eye-tracking technology, empirical researchers now have a very effective and unprecedented tool.

4. Conclusions

There are many open and un-investigated topics concerning program comprehension. We advocate for additional fundamental research on two topics: the scope of the understanding necessary for a given task and for improved methods for measuring the effectiveness of techniques. As needed comprehension versus understanding everything is a critical factor in the development of useful tools for developers. Eye-tracking equipment gives us a quantitative measure of comprehension and should assist in the development of better techniques and visualizations to support program comprehension.

A number of application areas were identified that could benefit from program comprehension research. The advent of generics and hard to understand generic libraries is one such area. The use of formal methods to support program comprehension is another area. Lastly, controlled experiments that present broader impacts on the computer science community, such as understanding differences in how people comprehend programs should be more widely undertaken.

5. References

[1] Beck, K., Extreme Programming Explained, Addison Wesley, 2000.

[2] Brooks, R., "Towards a theory of the cognitive processes in computer programming", International Journal of Man-Machine Studies, vol. 9, no. 6, 1977, pp. 737-742.

[3] Brooks, R., "Towards a Theory of the Comprehension of Computer Programs", International Journal of Man-Machine Studies, vol. 18, no. 6, 1983, pp. 543-554.

[4] Fisher, M., Cox, A., and Zhao, L., "Using Sex Differences to Link Spatial Cognition and Program Comprehension", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, PA, Sept. 2006, pp. 289-298.

[5] Fix, V., Wiedenbeck, S., and Scholtz, J., "Mental representations of programs by novices and experts", in Proceedings of Conference on Human Factors and Computing

Systems (INTERCHI'93), Amsterdam The Netherlands, April 24-29 1993, pp. 74-79.

[6] Hyona, J., Radach, R., and Deubel, H., The Mind's Eye:Cognitive and Applied Aspects of Eye Movement Research, Amsterdam, 2003.

[7] Jacob, R. J. K., "What you look at is what you get: eye movement-based interaction techniques", in Proceedings of SIGCHI conference on Human factors in computing systems: Empowering people, Seattle, Washington, United States, 1990, pp. 11-18.

[8] Koenemann, J. and Robertson, S., "Expert Problem Solving Strategies for Problem Comprehension", in Proceedings of Conference on Human Factors and Computing Systems (CHI'91), New Orleans, LA, April 27 - May 2 1991, pp. 125-130.

[9] Letovsky, S., "Cognitive Processes in Program Comprehension", in Empirical Studies of Programmers, Albex, 1986, pp. 58-79.

[10] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension", IEEE Software, vol. 19, no. 3, May 1986, pp. 41 - 48.

[11] Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E., "Mental Models and Software Maintenance", in Empirical Studies of Programmers, Albex, 1986, pp. 80 - 98.

[12] Pennington, N., "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", Cognitive Psychology, vol. 19, 1987, pp. 295-341.

[13] Purchase, H., C., Colpoys, L., McGill, M., Carrington, D., and Britton, C., "UML Class Diagram Syntax: An Empirical Study of Comprehension", in Proceedings of Asia-Pacific Symposium on Information Visualisation, Sydney, Australia, 2001, pp. 113-120.

[14] Ricca, F., Penta, M. D., Torchiano, M., Tonella, P., and Ceccato, M., "The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes", in Proceedings of 29th International Conference on Software Engineering (ICSE'01), 2007, pp. 375-384.

[15] Shneiderman, B., Software Psychology, Winthrop Publishers Inc., 1980.

[16] Smith, D., R., "Comprehension by Derivation", in Proceedings of IEEE International Workshop on Program Comprehension, St. Louis, MO, USA, May 15-16 2005, pp. 3-9.

[17] Soloway, E., Adelson, B., and Ehrlich, K., "Knowledge and Processes in the Comprehension of Computer Programs", in The Nature of Expertise, Chi, M., Glaser, R., and Farr, M., Eds., A. Lawrence Erlbaum Associates, 1988, pp. 129 - 152.

[18] Soloway, E., Bonar, J., and Ehrlich, K., "Cognitive Strategies and Looping Constructs: An Empirical Study", Communications of the ACM, vol. 26, no. 11, November 1983.

[19] Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge", IEEE Transactions on Software Engineering, vol. 10, no. 5, September 1984, pp. 595-609.

[20] Storey, M.-A., "Theories, Methods and Tools in Program Comprehension: Past, Present and Future", in Proceedings of IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, MO, USA, May 15-16 2005, pp. 181-194.

[21] Storey, M.-A., "Theories, Methods and Tools in Program Comprehension: Past, Present and Future", Software Quality Journal, vol. 14, no. 3, September 2006, pp. 187-208.

[22] Sutton, A. and Maletic, J. I., "Automatically Identifying C++0x Concepts in Function Templates", in Proceedings of 24th IEEE International Conference on Software Maintenance (ICSM'08), Beijing, China, Sept. 28-Oct. 4 2008, pp. (10 pages to appear).

[23] Uwano, H., Nakamura, M., Monden, A., and Matsumoto, K., "Analyzing individual performance of source code review using reviewers' eye movement", in Proceedings of 2006 symposium on Eye tracking research & applications (ETRA), San Diego, California, 2006, pp. 133-140.

[24] Von Mayrhauser, A. and Lang, S., "On the Role of Static Analysis during Software Maintenance", in Proceedings of Seventh International Workshop on Program Comprehension (IWPC98), Pittsburgh, Pennsylvania, 5 - 7 May 1998.

[25] Von Mayrhauser, A. and Vans, A. M., "Program Understanding - A Survey", Department of Computer Science, Colorado State University, Technical Report CS-94-120, August 23 1994.

[26] Von Mayrhauser, A. and Vans, A. M., "Program Comprehension During Software Maintenance and Evolution", Computer, vol. 28, no. 8, 1995, pp. 44-55.

[27] Von Mayrhauser, A. and Vans, A. M., "Program understanding behavior during debugging of large scale software", in Proceedings of Seventh workshop on Empirical studies of programmers, 1997, pp. 157–179.

[28] Woods, S. and Yang, Q., "The Program Understanding Problem: Analysis and A Heuristic Approach", in Proceedings of ICSE, 1996, pp. 6-15.

[29] Yusuf, S., Kagdi, H., and Maletic, J. I., "Assessing the Comprehension of UML Class Diagrams via Eye Tracking", in Proceedings of International Conference on Program Comprehension (ICPC'07), 2007, pp. 113-122.