Mapping a Sequence Diagram to the Related Code: Cognitive Levels Expressed by Developers

David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper
Digital Ecosystems and Business Intelligence Institute
Curtin University of Technology
GPO Box U1987
Perth WA 6845, Australia
{D.McMeekin, B.vonKonsky, E.Chang, David.Cooper}@curtin.edu.au

***Abstract*— This paper reports on a study in which developers' cognitive levels were categorised and measured while they mapped a sequence diagram to the related code based on a Usage Based Reading scenario. Results indicate that applying the usage-based reading technique to map a sequence diagram to the underlying code, facilitates a developer to operate at the Knowledge and Comprehension levels of Bloom's cognitive taxonomy, but does not facilitate sustaining it at the analysis level. The results of this study highlight the need for improved tools and methodologies that aid developers understanding of the system, particularly for those commencing a new project.**

I. Introduction

Complex software systems have become integral to most aspects of life in today's society. Products, services and infrastructure are designed, produced and delivered through the use of these software systems [6], [15].

Modifying and maintaining these systems is a complicated, time consuming process within the software development life-cycle. To do this successfully, the developer must understand the system so their changes achieve the given purpose without breaking the system in an unexpected location.

A software product spends almost 70% of its life span in the maintenance cycle and between 50% and 90% of this time is used by developers trying to understand the program [7].

With the shortage of experienced software engineers and developers along with the increased demand for software engineers/developers across many disciplines [6], it is important that new techniques and frameworks are created to assist developers new to a project, whether they are novice or experienced developers, to increase their understanding of the system they are working on in shortened periods.

This paper reports results from a study where developers performed a Usage-Based Reading (UBR) [28] inspection mapping a sequence diagram scenario to the related code. Participants were required to think aloud during the study [9], in order for the developers' different cognitive levels to be categorised and measured using a modified version of the Context-Aware Analysis Scheme for Bloom's Taxonomy [14]. This facilitated observation of the differing cognitive levels expressed by the developers as they performed the task.

This work observed the cognitive levels developers expressed while performing a task to map functionality described in a sequence diagram to the underlying code. It was not about how developers understand, or increase their understanding of, a system. The study's aim was to observe if performing a Usage-Based Reading (UBR) inspection of a sequence diagram enabled developers to operate at higher cognition levels as classified using Bloom's taxonomy.

II. Background

A. Software Inspections

Traditionally, software inspections have been used to detect defects within the product. Empirical research reports inspections have been very successful with the removal of up to 80% of defects prior to first release [10], [11]. It has also been reported that there is a correlation between how many defects an inspector detects and their ability to successfully add new functionality to that code [23]. Other research has shown that inspections make code easier to understand and change [27] and also facilitate developer operation at higher cognition levels during the inspection [20].

The Usage-Based Reading (UBR) [28] technique is an inspection technique that evolved from the Statistical Usage Inspection (SUI) [24]. The SUI technique attempted to certify a product's reliability by testing it in accordance with the expected usage. The UBR technique builds on this technique in that all defects are not considered equal. Defects that will have the most destructive impact on the system, from a user's perspective, are considered the most important. These are the defects that need to be detected and fixed as early in the development life-cycle as possible.

Prior to inspection, use case scenarios are prioritised and the inspector works through these scenarios, starting with the scenario that has the highest priority. The inspector systematically traces the scenario through the artefact under inspection ensuring that all needed functionality exists and is correct. In this manner, the prioritisation is designed to catch those defects that most affect the system's usability.

B. Program Comprehension

Comprehension is defined as "the capability of understanding something" [8] while understanding is defined as "perceiving the intended meaning of something" [8].

TABLE I
PROGRAM COMPREHENSION COGNITIVE MODELS.

Model	Author	Description
Bottom-up	Shneiderman [26]	"chunks" of code are created and grouped together until the problem has a solution.
Top-down	Brooks [4]	a global hypothesis describing the whole program is created. Further hypothesis refinement occurs and is tested until the program, in its entirety, is understood (this methodology is also described by Polya [25] and Wickelgren [29] described this problem solving methodology)
Systematic	Littman *et al.*[16]	a developer systematically reads through the software building their understanding by looking at data and control flow.
As needed	Littman *et al.*[16]	as the name indicates, a developer looks only at issues requiring immediate attention needed for the task at hand.
Integrated	Mayrhauser and Vans [18]	here a developer uses both the top-down and bottom up methodologies to best assist them in their understanding of the software

Program comprehension is an extremely important aspect in maintaining and evolving software systems. Software engineers need to gain an understanding of code they are unfamiliar with [17]. Several cognitive models exist that describe how developers build their comprehension of a software system's operational and functional behaviours. Table I cites and briefly describes 5 different models.

The cognitive models used to explain developer comprehension usually highlight 2 ways in which a developer's comprehension is acquired, either the build up of knowledge through the programmer studying what tasks the program performs (functional) or how the program performs those tasks (control flow) [17]. As a software developer embarks on maintaining and evolving a system, understanding both the functional and control flow of the system is of the utmost importance. Without them, the developer may make changes that appear correct in one location but introduce defects into the system in a different location.

C. Bloom's Taxonomy

Blooms taxonomy is a classification taxonomy that identifies different cognitive levels potentially exhibited during learning. The taxonomy has been widely embraced and used within educational disciplines [1], [3]. The six categories, cited from Bloom [3], are listed and briefly described, with an example of how each might be expressed in a programmers context:

- **Knowledge**: "retrieving relevant knowledge from long-term memory." For a programmer, this may be demonstrated via recollection of a for loop pattern.
- **Comprehension**: "construct meaning from instructional messages, including oral, written, and graphic communication." In programming, this may be summarising a code fragment's task.
- **Application**: "carry out or use a procedure in the given situation." For example, where the developer is making a change in the code.
- **Analysis**: "break material into constituent parts." In programming this may be demonstrated by describing how a field or method operates and its role within the wider program.
- **Evaluation**: "make judgements on criteria and standards". In this case, the developer may make an assessment of the way a program solves a specific problem.
- **Synthesis**: "reorganise elements into a new pattern or structure." A programmer creating a new method, adding new functionality would represent synthesis.

Bloom's taxonomy has been proposed as a way in which developers' cognitive levels can be assessed during different development tasks [5]. Results from different studies have been reported in [14], [30], [31], [32]. Moreover, a Context-Aware Schema has been proposed for applying Bloom's taxonomy [14]. In earlier studies, we have applied the schema and reported on the outcomes [20], [21], [22].

The think aloud data collection method, also known as Protocol Analysis [9], requires participants to verbalise their thoughts and actions as they execute a given task. The verbalisations are recorded and then used in the data analysis. Protocol analysis has been widely used within studies examining participants' cognitive levels expressed while carrying out a given task [2], [4], [12], [16], [19].

III. METHODOLOGY

In this study participants were given the task to map a use case scenario, shown in a sequence diagram, to the corresponding code. The different cognitive levels developers expressed while carrying out the task were observed. This data was then qualitatively examined. The use of a sequence diagram required participants to map functionality from the scenario and sequence diagram to the underlying code, hence employing a top–down comprehension strategy.

The study collected data in two ways:

1) think aloud data was collected from participants, and
2) an online interface was used to complete the scenario mapping to the underlying code.

The online interface data provided access to the way in which participants mapped the scenario to the code. From the interface it was possible to observe the steps taken by the participants as they executed the task. The think aloud data provided access to identifying the different cognitive levels, from Bloom's taxonomy, participants expressed during the process.

The software used in this study was a command–line controlled, Java–based audio player. The software contained seven Java classes consisting of 330 lines of code. A UML sequence diagram scenario was presented representing what happened

when a user selected the system's "next track" option while playing tracks in a random order.

A total of 10 participants took part in the study. Due to equipment failure, think aloud data from four participants was not intelligible and needed to be discarded. From the remaining six, two had industry experience and the remaining four were final year undergraduate students enrolled in either Computer Science or Software Engineering.

Participants had not seen the artefacts prior to commencing the task. Before starting the study, participants took part in a training exercise to practice and become familiar with using the think aloud protocol.

Participants performed an inspection–type task in which they were required to map the scenario in the sequence diagram to the underlying code. Participants needed to list the execution order of the lines of code that were or may have been executed if the scenario was run. Participants were given as much time as they needed to complete the study.

In the Context-Aware Schema [14] examples, the Analysis level is identified when a participant discussed code in relation to an external system. We differ from [14] in our interpretation of the Analysis level at this point. In our study when participants' utterances described code within the system but in different classes from that which they were currently examining, these utterances were categorised as Analysis level. Our reason for this is, this type of identification demonstrated an understanding of constituent parts and also a detection of relationship of parts of the system and the way they were organised [3].

A model solution for mapping the sequence diagram to the corresponding code was independently created by 2 of the researchers. Where differences arose between the 2 researchers solutions, consultations were held until the differences were settled. Once this solution was created, an external domain expert was consulted in order to verify the model solution.

Empirical research is subject to 2 types of validity threats, internal and external. Selection threat was the first internal threat faced by this study. Selection threat is where participants are selected in an attempt to produce favourable outcomes for the research. To counter this threat, an open invitation was made to graduate and final year undergraduate students. Students were selected on a first-come first-served basis, and once the full number of students was reached, the study was closed. It is possible that only the ambitious students volunteered to participate within this study, hence this must be considered when examining the results. Industry participants were invited via invitation to the companies which had expressed an interest in participating in our empirical research.

The second internal validity threat was that of participant experience. Demographics were recorded from each participant in an attempt to control this variable.

An external threat to validity within this study was that of sample size. Results from this study are not intended for generalisation, but to gain a qualitative understanding of the cognitive levels expressed by developers while performing the task at hand.

TABLE II
CORRECTNESS OF PARTICIPANTS SOLUTIONS

Participant	**% of model solution covered**	**% of mismatches**
1	42%	74%
2	85%	37%
3	33%	83%
4	63%	25%
5	62%	20%
6	73%	27%
Mean	**60%**	**44%**

IV. RESULTS

The think aloud data was collected, transcribed and then broken into sentences/utterances. A set of 150 utterances were coded by two independent researchers, using the Context–Aware Schema. An expert, in applying Bloom's taxonomy was also consulted on the Schema's application to the utterances. The Cohen's Kappa statistic was calculated to determine the agreement level between the two researchers' categorisations. The Kappa statistic was 0.63, which is considered a substantial agreement [13]. The uncoded utterances were then coded by a single researcher.

Figure 1 displays a break down of participants' utterances (p1–p6 and the mean) while carrying out the task. The graph shows that more than 70% of the participants' utterances are in the Knowledge and Comprehension levels. This makes up a significant portion of the participant's total time for the exercise. As no participant expressed an utterance that was either in the Application nor Synthesis levels during the task, these levels were omitted from the graph.

Figure 2 displays the different cognitive levels participant six expressed throughout the course of the exercise. The vertical axis reflects the different levels of Bloom's taxonomy. Participant six's graph was arbitrarily chosen as an example. This figure highlights the different cognitive levels the participant expressed as they mapped the sequence diagram to the code. From the graph it is seen that they operated for the vast majority of the time at the Knowledge and Comprehension levels and then for small time periods they would move into the Analysis level.

Looking at Figure 1 and Figure 2 it can be seen that the participants operated at the Knowledge and Comprehension levels for the majority of the time. They moved into the Analysis level at different points along the time line, and on occasions also operated at the Evaluation level.

No participant operated at the Synthesis level nor the Application level during the study. In the context of this study, mapping a sequence diagram scenario to the related code, it was expected that the participants would not operate at the Application or Synthesis level because the exercise did not require them to perform any task that required them to operate at those levels. The Synthesis level requires the creation of something new and the Application level requires the implementation, modification or evolution of some section

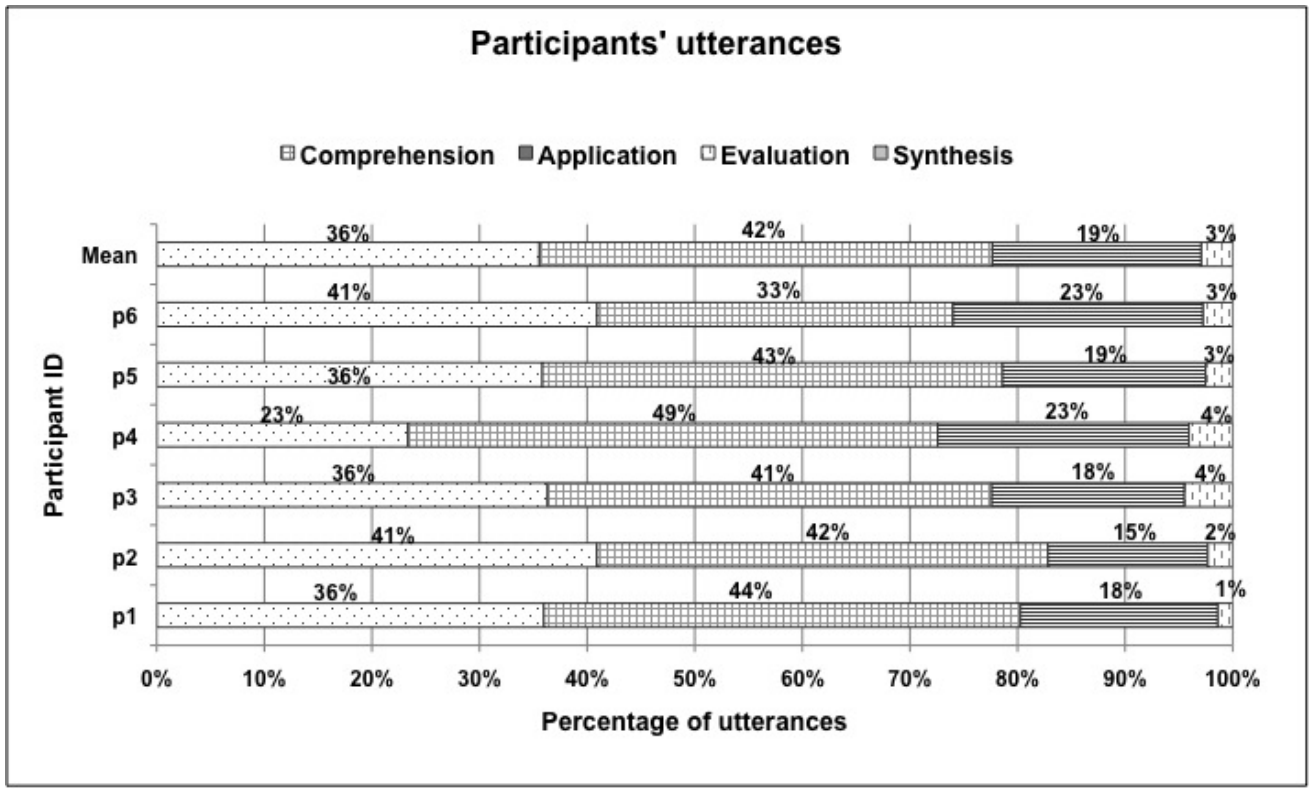

Fig. 1. A break down into Bloom's cognitive levels of participants' utterances.

of code. Hence, no participants functioned at those cognitive levels.

Table II characterises the correctness of the participants' solutions when compared to the model solution. These results show a great variety in the participants' solutions in comparison to the model solution, varying from 33%–85%. The table also shows that participants incorrectly mapped between 20% and 83% of executing lines of code with the model solution.

In examining the cognitive levels expressed and the results of mapping the corresponding code from the sequence diagram, simply functioning at the Knowledge and Comprehension levels did not correspond to correctness of solution. This may have arisen from the fact that participants were unfamiliar with the code base and therefore their time and effort was actually spent on attempting to gather the knowledge and understanding of the code prior to being able to successfully map the diagram to the code.

The task to map the sequence diagram to the underlying code requires the participant to implement a top–down cognitive strategy, mapping functionality to the underlying code. Table II shows that participants were not very successful in this task. This type of task strongly reflects what a person should be doing when functioning at Bloom's Analysis level. Figure 1 shows that participants did not function for very long at the Analysis level and the results show that they were not successful at completing the task.

V. Discussion

Participants were completely unfamiliar with the system in question. First and foremost they were becoming associated with the system. This can be seen in that the vast majority of utterances were in the knowledge and comprehension cognitive levels. Operating at these levels is equivalent to making oneself familiar with the system at hand.

As can be seen in Figure 2, (this graph is reflective of other participants' graphs), participants did not move from the lower cognitive levels steadily into the higher levels but in fact fluctuated between the Knowledge, Comprehension and Analysis levels.

Speculatively, it might have been beneficial for the task to have started with a generic familiarisation methodology, so that participants could build a basic understanding of the system, before requiring them to perform the sequence diagram mapping exercise. By separating the tasks, the initial build up of basic system knowledge and understanding, this may have facilitated participants to operate at the higher cognitive levels such as Analysis for longer and more sustained time periods.

An issue noted in the data analysis was that no participant successfully identified the correct starting point of the scenario in comparison to the model solution. This caused problems for the participants because to identify if a line of code executed or not one must first have identified all previous lines of execution. If previous lines were not successfully identified

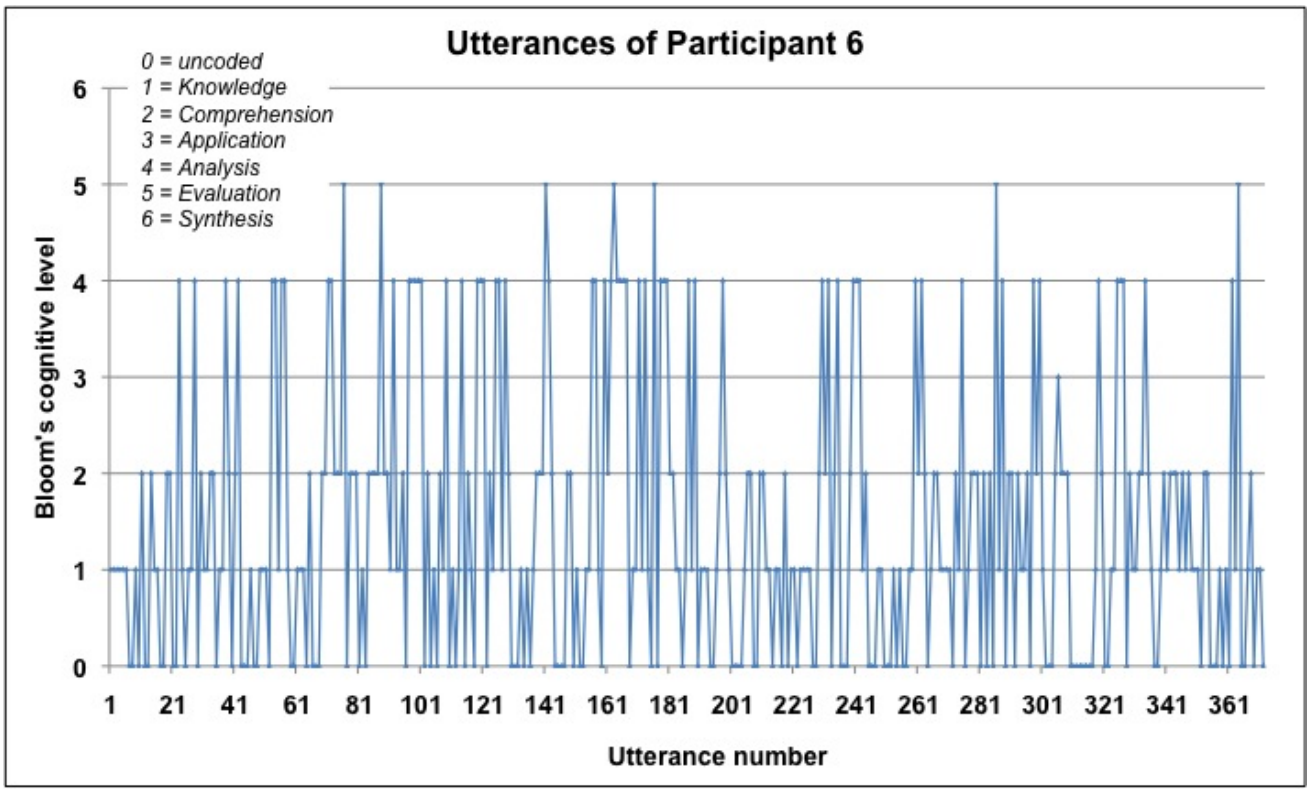

Fig. 2. A sample of one participant's utterances.

this posed a challenge for the participant to know if the line they were currently looking at would be executed or not.

This error may be reflective again of the fact that participants did not frequently function at Bloom's Analysis level. Successfully identifying the scenario's starting point would have indicated participants' operating at that Analysis level. The data analysis indicates that, on average, only 19% of the utterances were in the Analysis level.

Another issue identified within the data analysis was the incorrect identification of polymorphism. Participants generally referred to the parent class when examining the code even though the descendant class was shown on the sequence diagram. Functioning at the Analysis level, breaking the material into its constituent parts would have meant that the developer was understanding the class's wider role within the program.

The data analysis also highlighted participants incorrect evaluation of conditional statements. This incorrect evaluation meant that participants went on to examine incorrect code. Again, functioning at the Analysis level, it would be expected that participants identified the mismatch between the scenario and the code they were examining.

VI. Conclusion

This paper has shown that applying UBR of sequence diagrams, while helpful in moving developers through the Knowledge to the Comprehension cognitive levels, was not successful in moving developers to function largely at the Analysis or higher cognitive levels.

The nature of a sequence diagram correlates well with Bloom's taxonomy's Analysis level. The Analysis level also requires "breaking the material into its constituent parts" and identifying how objects interact and communicate with other objects within the system. It is also the mapping of high-level functionality down to the low-level code. For this reason it was expected that participants would have functioned at the Analysis level.

Reflecting on the task given to participants, it may be that the task given to participants to perform facilitated the Knowledge and Comprehension levels to have the highest counts. Requiring participants to map the sequence diagram to the code, with the understanding that upon completion of that task they would be required to add new or modify existing functionality within the system may have facilitated higher cognitive levels to be expressed. In this manner the purpose of studying the code would have been to perform a later task rather than simply mapping the sequence diagram to the code.

Using sequence diagrams to increase developer understanding of the system when they have no prior system knowledge may not be the most effective way to build their understanding of the system and code. It may have been that, had developers had a working knowledge of the system first, performing the sequence diagram to code mapping with another task, such as adding or modifying functionality in that area of the system, would have enable them to operate at the higher cognitive levels for longer and more sustained time periods of the task.

Increasing developer's cognitive understanding is essential to continue to increase the quality and safety of software. Without this cognitive system understanding, developers will continue to produce low quality code that may result in systems that are unsafe for use. Software is ubiquitous and pervasive in today's society, the chance that low quality software may endanger people's lives continues to increase.

Therefore, with demand high for software developers in most areas of business, and the apparent shortage of developers to take up these positions, it is of the utmost importance that new methodologies and frameworks be created to aid developers, either novice or experienced, new to a project, to better understand the system they are working upon in shorter time periods.

The conjecture from the results is that when building tools and/or methodologies for increasing programmer understanding, the combination of more than one task is needed to facilitate a developer's, whether novice or experienced, cognitive level moving from the lower to the higher levels.

REFERENCES

[1] L. W. Anderson, D. R. Krathwohl, and B. S. Bloom. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2001.

[2] D. Bergantz and J. Hassell. Information relationships in PROLOG programs: how do programmerscomprehend functionality? *Int. J. Man-Mach. Stud.*, 35(3):313–328, 1991.

[3] B. S. Bloom, editor. *Taxonomy of Educational Objectives Cognitive Domain*. David McKay Company, Inc., 1956.

[4] R. Brooks. Towards a theory of the comprehension of computer programs. *International Jouranl of Man–Machine Studies*, 18(6):543–554, 1983.

[5] J. Buckley and C. Exton. Bloom's taxonomy: a framework for assessing programmers' knowledge of software systems. *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 165–174, 5 2003.

[6] The Challenges of Complex IT Projects. Technical report, 2004.

[7] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of Fourth Workshop on Program Comprehension, 1996.*, pages 9–18, 3 1996.

[8] *Oxford English Dictionary*. Oxford University Press, 2008.

[9] K. A. Ericsson and H. A. Simon. *Protocol Analysis*. The MIT Press, 1993.

[10] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 3 1976.

[11] M. E. Fagan. Advances in Software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 7 1986.

[12] C. Fisher. Advancing the study of programming with computer-aided protocol analysis. pages 198–216, 1987.

[13] D. P. Hartmann. Considerations in the choice of interobserver reliability estimates. *Journal of Applied Behavior Analysis*, 10(1):103, 1977.

[14] T. Kelly and J. Buckley. A Context-Aware Analysis Scheme for Bloom's Taxonomy. In *14th IEEE International Conference on Program Comprehension, 2006, ICPC 2006.*, pages 275–284, 2006.

[15] S. C. Kothari. Scalable Program Comprehension for Analyzing Complex Defects. In *The 16th IEEE International Conference on Program Comprehension, 2008, ICPC 2008.*, pages 3–4, 6 2008.

[16] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 12 1987.

[17] A. v. Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings. ICSE-16.16th International Conference on Software Engineering, 1994.*, pages 39–48, 1994.

[18] A. v. Mayrhauser and A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[19] A. v. Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *Software Engineering, IEEE Transactions on*, 22(6):424–437, 6 1996.

[20] D. A. McMeekin, B. R. Konsky, E. Chang, and D. J. A. Cooper. Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008*, pages 222–227, 2008.

[21] D. A. McMeekin, B. v. Konsky, E. Chang, and D. J. A. Cooper. Measuring Cognition Levels in Collaborative Processes for Software Engineering Code Inspections. In *Innaugural ICST IT Revolutions Conference*, 2008.

[22] D. A. McMeekin, B. von Konsky, E. Chang, and D. J. A. Cooper. Evaluating Software Inspection Cognition Levels Using Blooms Taxonomy. In *IEEE 22 nd Conference on Software Engineering Education and Training, 2009, CSEET '09.*, 2 2009.

[23] D. A. McMeekin, B. R. von Konsky, E. Chang, and D. J. A. Cooper. Checklist Based Reading's Influence on a Developer's Understanding. In *Proc. 19th Australian Conference on Software Engineering ASWEC 2008*, pages 489–496, 2008.

[24] M. Olofsson and M. Wennberg. *New Reference*. PhD thesis, Department. of Communication Systems, Lund Institute of Technology and Ericsson Telecom AB, 1996.

[25] G. Polya. *How to solve it*. Doubleday, 1957.

[26] B. Shneiderman. Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, 9:465–478, 7 1977.

[27] H. Siy and L. Votta. Does the modern code inspection have value? In *Proceedings IEEE International Conference on Software Maintenance, 2001.*, pages 281–289, 2001.

[28] T. Thelin, P. Runeson, and C. Wohlin. An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering*, 29(8):687–704, 8 2003.

[29] W. Wickelgren. *How to solve problems*. W.H. Freeman, 1974.

[30] S. Xu and V. Rajlich. Cognitive process during program debugging. In V. Rajlich, editor, *Proc. Third IEEE International Conference on Cognitive Informatics*, pages 176–182, 2004.

[31] S. Xu and V. Rajlich. Dialog-based protocol: an empirical research method for cognitive activities in software engineering. In V. Rajlich, editor, *Proc. International Symposium on Empirical Software Engineering*, pages 10 pp.–, 2005.

[32] S. Xu, V. Rajlich, and A. Marcus. An empirical study of programmer learning during incremental software development. In V. Rajlich, editor, *Proc. Fourth IEEE Conference on Cognitive Informatics (ICCI 2005)*, pages 340–349, 2005.