Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension

David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper
Curtin University of Technology
Digital Ecosystems and Business Intelligence Institute
Kent Street Bentley, W.A. 6102, Australia.
D.McMeekin@curtin.edu.au

Abstract

Software maintenance can consume up to 70% of the effort spent on a software project, with more than half of this devoted to understanding the system. Performing a software inspection is expected to contribute to comprehension of the software. The question is: at what cognition levels do novice developers operate during a Checklist-Based code inspection followed by a code modification? This paper reports on a pilot study of Bloom's taxonomy levels observed during a Checklist-Based inspection and while adding new functionality unrelated to the defects detected. Bloom's taxonomy was used to categorise think-aloud data recorded while performing these activities. Results show the Checklist-Based Reading technique facilitates inspectors to function at the highest cognitive level within the taxonomy and indicates that using inspections with novice developers to improve cognition and understanding may assist integrating developers into existing project teams.

1. Introduction

Software inspections have been used to reduce the defect count in software. Empirical research has shown more than 80% of defects can be removed and programmer time saved by implementing software inspections [8, 9]. The Checklist-Based Reading (CBR) technique remains the dominant inspection methodology used in industry [23].

Development methodologies, tools, technology, and implementation languages have evolved independently of software inspections. The evolution of strongly typed languages, automated tools, and significantly improved testing environments has resulted in many defect groupings becoming extinct [20].

This paper looks at applying software inspections into the developer comprehension domain rather than into their traditional defect detection domain. It reports on results from a study using the context-aware analysis scheme [14] to measure comprehension levels during a code inspection followed by a modification to that code requiring developers to add new functionality. We included a sixth category in the context-aware schema. This study used Bloom's taxonomy to examine the cognition level of a student/novice programmer while (1) performing a CBR inspection, and (2) adding new functionality to the inspected code.

Applying inspections in this non-traditional area may show cost-effective ways that developer comprehension can be improved. The results from this study will assist in answering the research question: at what cognition levels do novice developers operate during a Checklist-Based code inspection followed by a code modification?

2. Background

Software inspection techniques are a series of steps that guide software inspectors as they search for defects within a software artefact. These techniques enable well-defined, systematic artefact inspection strategies that provide both feedback and improvement [19]. Performing an inspection on code prior to adding functionality was shown to improve developer ability to make required code modifications [16].

In a CBR inspection, the inspector asks a series of questions regarding the code. Each question requires a 'yes' or 'no' answer. A 'yes' indicates no defect while a 'no' answer indicates there may be a defect [8]. The questions that form a checklist are created from historical data collected from earlier defects experienced within the organisation [11, 12]. The checklist technique is a structured systematic methodology for inspecting software artefacts. For the novice, this technique guides them through the inspection. However, some experienced developers have found this methodology to be limiting, as it did not allow for their own experience to be incorporated into the inspection pro-

978-0-7695-3176-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPC.2008.20

cess (recently unpublished work conducted by the authors).

Models have been developed that attempt to explain the manner in which software developers build their understanding of the code [3, 15, 18, 26]. More recently, Bloom's taxonomy has been proposed as a framework to assess the programmer's knowledge of a software system [4] and further developed with a context-aware analysis scheme for the taxonomy [14].

Kelly and Buckley [14] state that two distinct tracks have evolved in research on software comprehension. The first, how software is represented and what impact these representations have upon comprehension level is split into two categories: (1) external representations, such as system source code and documentation, and (2) internal representations such as programming plans [21] and beacons [27]. The second is in the cognitive processes domain that developers implement as they attempt to comprehend the systems they are working on.

Table 1 shows five different cognitive models that have been suggested within the literature.

Educators lead by Bloom developed a classification taxonomy that identifies different cognition levels in learning. This has become widely adopted and used within educational circles [2]. This taxonomy became known as "Bloom's Taxonomy of Educational Objectives." Six cognitive groups were created ranging from low to high cognitive levels:

- **Knowledge**: is the recalling or remembering of facts, specifics, or patterns. In programming, this may be demonstrated by the recalling of while loop patterns.
- **Comprehension**: is where one understands concepts and knowledge and can explain it to others in a different manner. For the programmer, this may be represented by summarising what a code fragment does.
- **Application**: is applying knowledge in a different situation to solve the current problem at hand. A programming example is where the developer is making a change to the code.
- **Analysis**: is where complex information is recognised to reveal hidden meaning. For example, a programmer demonstrates analysis by describing how a field or method operates and its role within the wider system.
- **Synthesis**: is where a new whole is formed through the combining of different elements and parts. For example, a programmer creating a new method that adds new functionality, not just modifying existing functionality in the code.
- **Evaluation**: is where judgements regarding the work are made. Here programmers may make an appraisal of the way in which a program solves a particular problem.

Bloom's Taxonomy has been proposed and used in studies examining the comprehension levels at which developers operate during different software engineering tasks. Buckley and Exton [4] proposed using Bloom's Taxonomy to characterise the various cognitive levels a programmer applies when executing different undertakings.

Xu and Rajlich [17, 28, 29, 30] used a Bloom's Taxonomy verb table to measure the different cognitive levels. They noted that their approach did not adequately measure the higher cognitive levels.

Kelly and Buckley [14] highlight two problems solved by a Context-Aware Schema for Bloom's Taxonomy. The first being that ambiguity is found because some verbs appear in more than one taxonomy level, and the second being reductionism, in that some data is not catered for and through its omission, valuable information may be lost.

This Context-Aware Schema [14] accounts for five of the six categories within Bloom's taxonomy. The Synthesis category was omitted because it requires the creation of something new, such as building a new program. Within their proposal's context, software maintenance, the building of something new does not occur, hence the justification for its removal.

The schema is based on sentence or utterance analysis. The schema allows for each utterance to be categorised based upon the two previous utterances. This means that sentences or utterances are examined in a wider context of usage to ensure correct categorisation [13].

Think-aloud is a process by which participants verbalise their thoughts and actions while performing a task [7]. This protocol has assisted in understanding how developers work and the cognitive processes they use as they work on software. Much of what is actually understood about the cognitive processes developers apply when performing different tasks is due to think-aloud analysis [1, 3, 5, 10, 15, 25].

3. Methodology

A CBR inspection was performed upon a single Java class from within a software system. The class contained 13 seeded defects. After completing the inspection, participants were presented with the same class, without the seeded defects, and a modification request for the inspected class. For the duration of the inspection and modification, participants were required to "think-aloud" and this was recorded via a microphone using the software Audacity.

CBR inspections are considered the de-facto industry standard. Thelin *et al.* [22] recommended when conducting empirical research involving inspections, CBR inspections be used as a baseline. Consequently, we chose to test the CBR inspection technique first.

In our application of the Context-Aware Schema [14], we re-introduced the synthesis category. This was because participants performed a modification to the code that added

Table 1. Cognitive Models Proposed within the Literature.

Model	Author	Description
Bottom-up	Shneiderman [18]	code "chunks" are grouped until a solution is created.
Top-down	Brooks [3]	a general hypothesis describing the entire program is created. This hypothesis is refined and tested until the program is understood
Systematic	Littman *et al.*[15]	a developer systematically reads through the software, building their understanding by looking at data and control flow.
As needed	Littman *et al.*[15]	looks only at code that needs immediate attention.
Integrated	Mayrhauser and Vans [26]	combines both the top-down and bottom-up methodologies

new functionality. This required the participants to create something new within the existing code and, hence, justified the synthesis level's inclusion into the data analysis.

The software artefacts used within this study were created specifically for use in this study. The system was a Java, text-based implementation of the board game BattleShip and consisted of seven classes. The game was designed for a single player to challenge the computer. The class inspected was the Board class, which contained 169 executable lines of code. This class represented the game board and was responsible for storing ships, receiving attacks, determining if a ship had been hit and determining if the game had been won. Participants were informed that the code compiled and executed. During the inspection, participants also had access to the Java API documentation.

Inspectors were presented the following artefacts (the checklist and defect recording sheet were paper-based while other artefacts were online):

1. natural language specification
2. a system class diagram
3. the Java code to be inspected
4. all other Java code in the system
5. checklist for guidance through the inspection process
6. defect recording sheet

To establish a baseline for participation, students needed to have successfully completed: two introductory Java courses and two introductory software engineering courses. Five participants (p1 - p5) took part in the study. Three were final year undergraduates enrolled in computer science and software engineering, one was a PhD student and the fifth was a recent graduate. Participation within the study was voluntary, not part of any course and no part of the material or learning done through the study was examinable within participants' degree programs.

Two small training exercises were conducted. The first was an inspection task, which was conducted to familiarise participants with the CBR inspection technique. Second, a "think aloud" exercise was conducted as shown in [7], to assist participants in using this technique.

The study first required each participant to perform a CBR inspection on a single class, followed by a modification of the code inspected. Participants were given 30 minutes to read the natural language specification and to complete the inspection. Each defect discovered was noted in the defect recording sheet with its line number and a description. No explanation as to how to fix the defect was needed.

Following the inspection, participants were given 30 minutes to add new functionality, unrelated to the seeded defects in the original code. Participants were given a defect free version of the Board class for this task. The new functionality was set out in an online Modification Request document and required an extension of the Board class, allowing players to place ships diagonally down to the right. The modification was conducted online with participants able to compile their code as often as they chose.

Thirteen defects were seeded within the inspected code based on prior research [6] and experience. For example, when an attack was made, the board location was to be labelled "BOMBED." However, it was labelled "EMPTY." This error was related to a developer using a similar piece of code and failing to change it from its original form. The incorrect labelling lead to the system failing, as it allowed the same location to be bombed multiple times.

Empirical research is subject to internal and external validity threats. Selection threat, the first internal threat, is where participant selection can be stacked to produce more favourable results. To limit this effect, an open invitation was made for student participation within the study and students who asked to participate and who met the baseline requirements, did. It is possible that only the ambitious, self-motivated students took part and this could affect the results.

The second internal threat, as with many software engineering studies, was participants' varying experience and background in the issues being studied. When using stu-

Table 2. Utterance Examples by Category

Bloom's Level	Utterance Example
Knowledge	String shipsAlreadyUsed numberOfShipsPlaced
Comprehension	we actually have a one to many relation
Application	in isTheSpaceFree we'll have to allow for a new direction.
Analysis	but board should have a ship
Synthesis	we are going to have to make a new one.
Evaluation	calls Board inBoard that is correct
Uncoded	Now I'll just check over the spec

dents as participants, it is possible they will be at different levels in their courses and have varying levels of industry experience. Demographic data was collected to assist with reducing this threat.

The sample population size posed an external validity threat to this study, as participating students may not be representative of the wider community. Results from this study are not planned to be generalised to the wider development community, but to indicate what may need to be examined in closer detail with a larger sample size in future work.

4. Results

The think-aloud recordings were transcribed and broken down into sentences, with each sentence considered an utterance. Two researchers coded a set of 130 utterances using the taxonomy schema and differences between the codings were discussed to clarify the schema's application. Cohen's Kappa was used to determine inter-observer reliability between researchers' utterance categorisations. The Cohen's Kappa was 0.605, which is considered acceptable, and the remaining utterances were coded by one researcher.

Table 2 gives a coded example utterance from each category within the context-aware schema. The "uncoded" category accounts for utterances that could not be categorised using Bloom's taxonomy. During the inspection and modification, an utterance was placed in this category when it did not relate to the inspector's current cognitive process. During the inspection, the uncoded listing also contained utterances when an inspector read a question from the checklist.

Table 3 and Table 4 show the breakdown of participants' utterances during the code inspection and modification respectively. Figures 1 and 2 show the utterance distribution for each of Bloom's categories during the inspection and modification respectively.

The Evaluation category in Figure 1 shows the median

Table 3. Inspection Utterances Summary

Bloom's Level	p1	p2	p3	p4	p5
Knowledge	38	91	32	42	34
Comprehension	22	62	19	41	25
Application	0	0	0	0	0
Analysis	12	23	23	16	16
Synthesis	0	0	0	0	0
Evaluation	31	56	79	86	54
Uncoded	27	51	86	86	78
Total	**130**	**283**	**239**	**271**	**207**

Table 4. Modification Utterances Summary

Bloom's Level	p1	p2	p3	p4	p5
Knowledge	27	39	18	21	17
Comprehension	5	16	4	10	9
Application	27	26	34	23	57
Analysis	4	11	4	4	11
Synthesis	12	6	9	6	16
Evaluation	6	16	10	7	11
Uncoded	13	30	27	36	72
Total	**94**	**144**	**106**	**107**	**193**

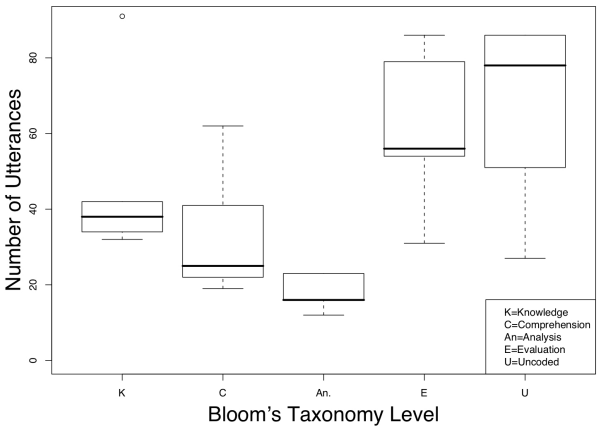

Figure 1. Number of utterances in each category during inspection

and range to be much higher than the other categories (the one category with a higher median is the Uncoded category). Each checklist question requires a yes or no answer. The inspector gives the answer based on their evaluation of the code, ideally resulting in them functioning within Bloom's Evaluation category. Correct implementation of a CBR inspection should lead the inspector to function at the Evaluation level. The Knowledge and Comprehension categories form the next two largest utterance blocks. Participants had not seen the code prior to inspection, so they needed to familiarise themselves with it. Based on the think-aloud data, inspectors were observed to have operated in these categories. The Analysis category has a small number of utterances within it. The inspection was on a single class within the system, hence inspectors may not have examined system-wide interactions in detail.

Figure 2 shows that, during the modification, the distribution spread for the Application category was the highest within Bloom's categories. Knowledge was the next highest with the remaining 4 categories all lying at similar places. By introducing the Synthesis category, a difficulty was noted on several occasions; it was difficult to distinguish these utterances from Application category utterances. For example, in creating the `setOceanDiagonal()` method, several participants copied and pasted either the `setOceanVertical()` or `setOceanHorizontal()` method. This was categorised in the Application category as it was reuse of that which already existed. However, changes needed to successfully implement the `setOceanDiagonal()` method required the creation of something new that used the knowledge, understanding and code already inspected. Therefore, utterances were categorised as Synthesis rather than Application, when they specifically used words such as "create" and "make a new," or if it was clear from the 2 previous utterances and the following utterance that it was the creation of something new.

5. Discussion

Participants three, four and five systematically followed the CBR inspection process, while participants one and two state in the think-aloud data that they deviated from the systematic implementation. Table 3 shows participants three, four and five operating more in the Evaluation category than participants one and two. Participants one and two operated largely in the Knowledge category during the inspection. Figure 1 shows the Analysis category with a small utterance distribution. A reason for this may be that, when using the CBR technique, inspectors familiarise themselves with the code, the Knowledge and Comprehension categories, and then start using the checklist and hence overlook the Analysis category. The Evaluation category distribution, shown on Figure 1, suggests that systematically applying a CBR inspection facilitates participants to operate in the Evaluation category of the taxonomy. Further study needs to be conducted in order to compare results between the CBR and another technique to validate this theorisation.

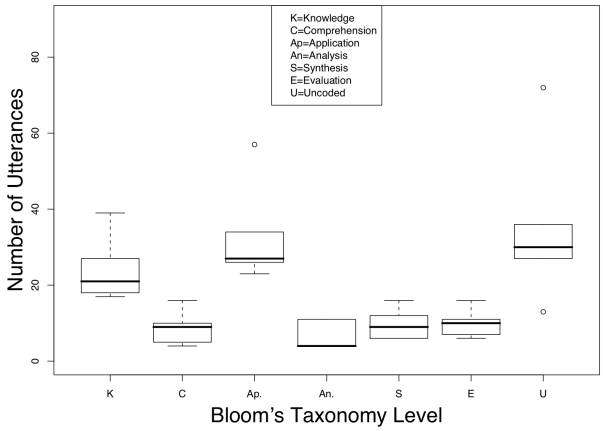

Figure 2. Number of utterances in each category during modification

Participant two's majority of utterances were in the Knowledge and Application categories. This may have resulted from the failure to execute the CBR inspection task correctly. When they reached the modification task, the participant still needed to increase their knowledge of the class until they were able to apply the requested modification.

6. Conclusion

This pilot study has shown that the context-aware schema for Bloom's taxonomy [14] is a means to measure the differing cognition levels at which a developer operates when performing CBR inspections followed by modifications. The Synthesis level was introduced for adding functionality to the inspected system. Results indicated that performing a CBR inspection facilitated inspectors operating at the highest cognition level, which may be due to the question and answer nature of CBR.

Results suggest that inspections may increase a recent graduate developer's overall cognition and understanding, hence aiding and facilitating their integration into existing project teams. Moreover, the improved system congnition is a benefit of utilising code inspections above and beyond their traditional use for defect detection.

Future work within this field will examine other code inspection techniques and the impact they have on a developer's comprehension level. A comparison of the context-aware analysis scheme, the lexical analysis scheme [29] and

the AFECS proposed in [24] will also be carried out, examining differences that may arise between the schemes.

References

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

[2] B. S. Bloom, editor. *Taxonomy of Educational Objectives: Book 1 Cognitive Domain*. David McKay Company, Inc., 1956.

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

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

[5] D. Cooper, B. von Konsky, M. Robey, and D. McMeekin. Obstacles to comprehension in usage based reading. *aswec*, pages 233–244, 2007.

[6] A. Dunsmore, M. Roper, and M. Wood. The development and evaluation of three diverse techniques for object-orientated code inspection. *IEEE Transactions on Software Engineering*, 29(8):677–686, Aug. 2003.

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

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

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

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

[11] T. Gilb and D. Graham. *Software Inspection*. Addison–Wesley, Wokingham, 1993.

[12] W. Humphrey. *A Discipline for Software Engineering*. Addison–Wesley, Massachusetts, 1995.

[13] T. Kelly. Bloom's taxonomy classification. Web page, 2008. *http://www.lit.ie/research/researchstaffpages/tarakelly.html* Last accessed on the 1st Feb. 2008.

[14] T. Kelly and J. Buckley. A context-aware analysis scheme for bloom's taxonomy. In *Proc. 14th IEEE International Conference on Program Comprehension ICPC '06*, pages 275–284, 2006.

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

[16] D. McMeekin, B. von Konsky, E. Chang, and D. Cooper. Checklist based readings influence on a developers understanding. In *Proc. 19th Australian Software Engineering Conference ASWEC '08*. IEEE Computer Society, 2008.

[17] V. Rajlich and S. Xu. Analogy of incremental program development and constructivist learning. In S. Xu, editor, *Proc. Second IEEE International Conference on Cognitive Informatics*, pages 98–105, 2003.

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

[19] F. Shull, I. Rus, and V. Basili. Improving software inspections by using reading techniques. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 726–727, Toronto, Ontario, Canada, 2001.

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

[21] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. pages 235–267, 1989.

[22] 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, Aug. 2003.

[23] C. K. Tyran and J. F. George. Improving software inspections with group process support. *Communications of the ACM*, 45(9):87–92, 2002.

[24] A. von Mayrhauser and S. Lang. A coding scheme to support systematic analysis of software comprehension. *Transactions on Software Engineering*, 25(4):526–540, 1999.

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

[26] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[27] S. Wiedenbeck. Beacons in computer program comprehension. *Int. J. Man-Mach. Stud.*, 25(6):697–709, 1986.

[28] 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.

[29] 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.

[30] 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.