Evaluating Software Inspection Cognition Levels Using Bloom's Taxonomy

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

Abstract

This paper reports on results from a pilot study that used Bloom's Taxonomy to observe cognition levels during software inspections conducted by undergraduate computer science and software engineering students. Cognition levels associated with three different code inspection techniques were investigated. These were the Ad hoc, Abstraction Driven, and Checklist-based reading strategies. Higher cognition levels were observed when using inspection techniques that utilise a more structured reading process. This result highlights the importance of introducing novice programmers to structured code reading strategies. Findings suggest that teaching different software inspection techniques throughout software courses, beginning with structured techniques, is an excellent way to build a student's critical software reading and analysis skills.

1. Introduction

Reading is an essential part of all educational disciplines. For example:

- writing students read the work of published novelists to understand and experience different writing styles and techniques;
- philosophy student read classical works to analyse structured arguments and propositions; and
- supervisors often suggest that Higher Degree by Research students read theses examples to assist them in writing their own dissertations.

Software development has been one of the few creative disciplines in which developers are generally not encouraged to read the works of their peers, or of more experienced developers [9]. However, the increased availability of Open Source software provides publicly available code that can be read and analysed by both students and developers [15].

Reading is fundamental to the production of high-quality software during both the development and maintenance life cycle of a software product [2]. However code inspection reading strategies are usually only taught in conjunction with verification and validation subjects.

Moreover, the main purpose of software inspections is to detect code defects. The use of inspections to raise design cognition and reading skills has not generally been acknowledged as a secondary benefit of the inspection process. Studying the use of code inspections for this purpose is the principal contribution of this study.

2. Background

Inspections present a structured method to examine code to identify defects [13].

978-0-7695-3539-5/09 $25.00 © 2009 IEEE
DOI 10.1109/CSEET.2009.15

There has been much written and researched in the application of inspections for detecting defects [14]. There is very little research as to how software inspections influence a developer's understanding of the artefacts being inspected.

The remaining parts of this section describe: several different inspection techniques used within this study, a summary of Bloom's taxonomy and its application within this study using the Context-Aware Schema [10].

2.1. Ad hoc reading

This technique is considered one of the simplest techniques used. The technique provides the inspector with no guidance regarding how to proceed. However, it is very effective. The inspector is expected to carry out a thorough systematic review based upon their personal experience [11].

A strength of this method is that it gives the experienced developer freedom to read the code as they like [4,11]. However, a weakness of this method is that a novice developer often does not have the experience needed to successfully apply this method.

2.2. Abstraction Driven Reading

The Abstraction Driven Reading (ADR) technique was described by Dunsmore *et al.* [5,6], and created specifically for Object-Orientated (OO) code. The inspector reads code systematically, writing an abstract natural language description about each method and then each class. While reading the code, if the inspector encounters delocalised code that is not found in the class, they trace the code execution out to those other classes. As they read the code and write the abstracts, ADR inspectors also compile a list of detected defects. It would be reasonable to expect that an inspector's understanding of the code is increased as they follow method calls leading them to code not directly covered by the inspection.

A strength of this method is that the natural language specifications created can be used for future inspections as well producing documentation for future reference. A weakness in this method is that it is very time consuming and the inspector can spend much time examining code outside the inspection's scope.

2.3. Checklist Based Reading

The Checklist Based Reading (CBR) technique was formalised by Fagan [8] and is the standard used in many software organizations throughout the world [11]. The inspector has a list of questions to ask about the artefact being inspected. Answering yes to a question indicates no defect, while answering no to a question indicates there may be a defect and a closer examination is needed.

A strength of this method is that it explicitly directs the inspector as they search for defects. This is very useful for a novice developer because of its explicit directions. Continuous and frequent application of this technique may assist the developer to gain experience regarding pitfalls and the common cause of defects.

A weakness in the method is that the attention of inexperienced developers can be directed away from defects not directly addressed by checklist questions.

2.4. Bloom's Taxonomy

Bloom's Taxonomy of educational objectives was created in 1956 [3], and revised in 2001 [1], to categorise different cognition levels developed during learning. Six different cognitive levels were identified ranging from high to low cognitive levels. This taxonomy has been widely used throughout the world within different education systems. The taxonomy consists of six cognitive categories briefly described below, with an example of its usage in a software developer's context:

Knowledge: "retrieving relevant knowledge from long-term memory."[1] In programming, this may be demonstrated by the recalling a specific control construct.

Comprehension: "Construct meaning from instructional messages, including oral, written, and graphic communication."[1] A programmer may summarise what task a code fragment performs.

Application: "carry out or use a procedure in the given situation."[1] A programmer may demonstrate this is by making a change in the code.

Analysis: "break material into constituent parts."[1] For example, a programmer describing how a field or method operates and its role within the wider system is analysis.

Synthesis: "re-organise elements into a new pattern or structure."[1] Here a programmer may create a new method adding new functionality to the code.

Evaluation: "make judgements based on criteria and standards."[1] Programmers demonstrate this by making an appraisal of the way in which a program solves a problem.

2.5. Context aware schema

Kelly and Buckley [10] proposed a context aware schema using Bloom's taxonomy to categorise developers' cognitive processes encountered while performing different software maintenance tasks. Their schema accounted for only five of the six categories, as their original proposal was in the software maintenance context. Consequently they removed the synthesis/creation category.

The schema requires developers to "think-aloud" [7]. Think-aloud is where participants verbalise their thoughts and actions while completing the task. In applying this scheme for analysis, data is divided into sentences or utterances. These sentences or utterances are categorised into one of Bloom's levels based upon their content as well as the two previous utterances. This allows for an utterance to be categorised in its wider context.

Thompson *et al.* [16] proposed using Bloom's taxonomy for programming assessment within the university context. The concrete examples given in that paper were very helpful when using Kelly and Buckley's [10] context aware schema to categorise "think-aloud" [7] data.

3. Methodology

An Ad hoc, ADR and CBR inspection were conducted on a single Java class belonging to a larger software system. The inspected class consisted of 169 effective lines of code and contained 13 seeded defects. In the context of this study: a defect was defined as "a deviation from specification that would go on to cause failure (some undesired behaviour of either a trivial or catastrophic nature) if left uncorrected" [4]. For the inspection, participants were required to "think-aloud." This data was recorded via microphone.

The software artefacts used were created specifically for this study. The software system was a text-based version of the Battleship Game. The Board class was inspected, and this was responsible for ship placements, attacks, and determining if a ship had been hit or not and was still floating or not.

Prior to participating in the study, students attended a two-hour lecture about software inspections. The lecture included the historical basis of inspections, empirical research results, as well as the examination of several different software inspection techniques.

Before starting the inspection, all participants were involved in a short training session to familiarise themselves with the specific inspection technique they were to implement and given exercises to practice the "think-aloud" protocol as shown in [7].

The study was advertised on campus. Students participated in their own time, were not paid for participation, and were informed that participation would have no influence on their marks in any subject within their degree program.

Participants were third year Software Engineering, Computer Science and Information Technology students. The participants conducted either: a CBR inspection, an ADR inspection, or Ad hoc inspection of a single Java class from within a larger software system.

Each participant was given the following artefacts: (1) A natural language specification, (2) A class diagram of the system, (3) The Java code to be inspected, (4) Access to all other Java code within the system, (5) A checklist (to those performing the CBR inspection), (6) A defect-recording sheet. All artefacts were online except the defect-recording sheet, which was paper based. Students were given 30 minutes to carry out the code inspection.

This kind of empirical research is subject to internal and external validity threats. The first encountered in this study was selection threat, where participant selection can be stacked to produce "better" results. In limiting this effect, we made an open invitation to final year students and those who responded participated.

The next internal threat was that of participant experience level. Considering this, only final year students participated and demographic data about each participant was collected to identify any discrepancies that may have arisen.

The external threat in this study is that of sample size. The think-aloud data collection method has significant overheads in collecting, collating, transcribing and analysing. The sample size was kept to 20 in order to validate that the research area may need to examined more in more detail. However, Moore and McCabe [12] point out that even with small sample sizes significant differences will still be identified.

4. Results

Participants were divided into three groups of five and each group used a different inspection technique. The think-aloud data was transcribed, and broken into utterances. Each utterance was categorised, using the Context-Aware Schema, into one of five of Bloom's levels. Table 1 lists an example utterance from each category. The Synthesis level was omitted in this study, as this category requires creating something new, for example, a new method or class. In this study participants performed an inspection and were not required to add anything new to the code. Hence, the Synthesis category was omitted.

Table 1. Example of utterances.

Bloom's level	Utterance Example
Knowledge	"while ship not placed"
Comprehension	"here is a one to many relationship"
Application	"here we must cater for a new direction"
Analysis	"this is controlled externally"
Evaluation	"the call here is wrong"

A set of 130 utterances was categorised by two different researchers and a Cohen's Kappa calculation performed to check for inter-observer reliability. The Kappa value was 0.605, which is considered an acceptable level. Consequently, a single researcher categorised the remaining utterances.

A statistical analysis was conducted using SPSSv16. A Kuskal-Wallis test was performed comparing each Bloom category with that same category for the three different inspection techniques. The test returned p-values of 0.004 and 0.006 for Knowledge and Evaluation respectively. This indicates a significant difference between the CBR technique and both Ad hoc and the ADR technique in these 2 areas.

Figure 1, 2 and 3 show the percentage breakdown, into Bloom's categories, of each student's utterances during the inspection, according to the inspection technique used. Uncoded utterances, ones that could not be coded into a Bloom category because they were unintelligible or unrelated to the task at hand, have been omitted.

Figure 1 shows students who implemented the Ad hoc inspection technique, on average had the highest percentage of utterances in the Knowledge. This is the lowest cognitive level in Bloom's taxonomy. These same students also had the lowest percentage of utterances in the Evaluation category, which is the highest cognitive level in Bloom's taxonomy considered in conjunction with this study.

Figure 2 shows those students who implemented the ADR inspection technique, on average, made fewer Knowledge utterances when compared to the utterances of the Ad hoc inspectors. These students also had a, on average, higher utterance count in Bloom's Evaluation level.

Figure 3 shows that students who carried out the CBR inspection had, on average, the lowest utterance count in the Knowledge category and the highest utterance count in the Evaluation category when compared with the students who used either the Ad hoc or ADR inspection techniques.

Participants, who used the CBR and Ad hoc inspection methods, on average, detected 2 defects each. Participants, who used the ADR method, on average, detected 3.5 defects each. In the ADR group, two participants, although final year students, had been working in industry for almost 12 months, and both on mission critical type systems. Hence, their experience level was much higher than other participants and they also detected more defects increasing the average defect count for the ADR group.

5. Discussion

The Ad hoc technique is the least structured of the three techniques examined in this study.

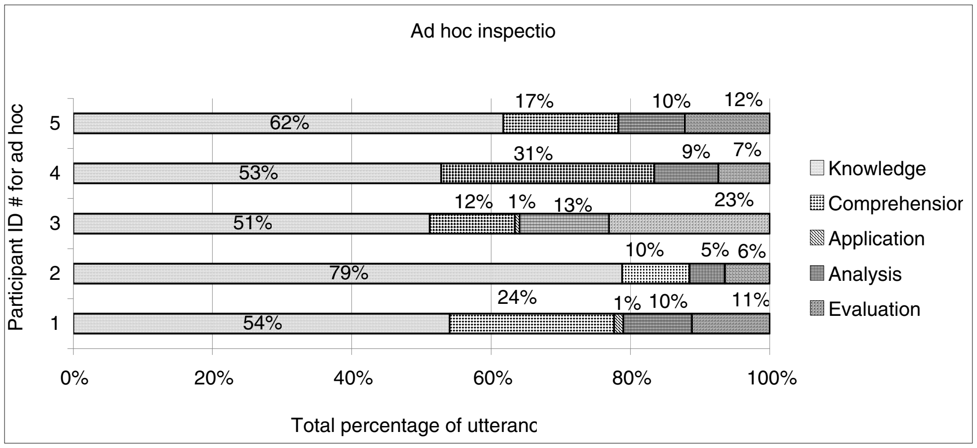

Figure 1. Bloom's categorization of Ad hoc utterances.

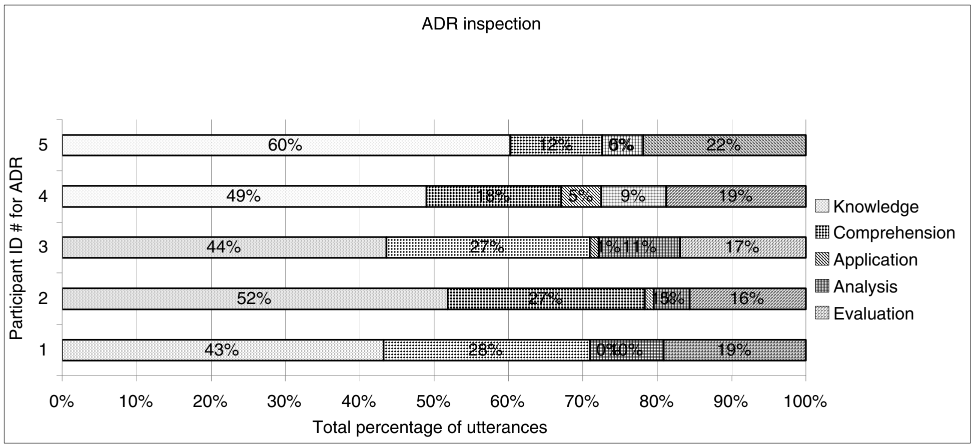

Figure 2. Bloom's categorization of ADR utterances.

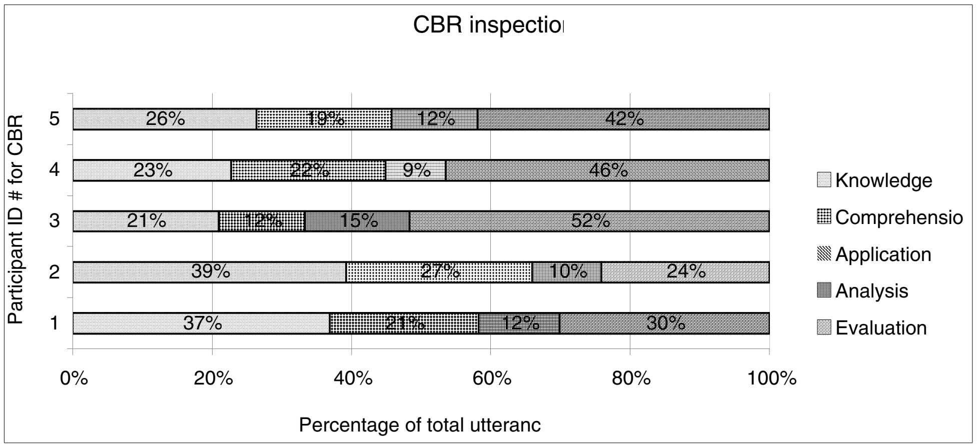

Figure 3. Bloom's categorization of CBR utterances.

It provides no instructions or direction regarding how to carry out the inspection. In this case, those students who conducted this style of inspection were simply given the task to find defects. The method did not describe how they should go about it, but were simply instructed to find the defects. With no additional guidance, students needed to rely on their own past experiences to carry out the task. Based on an analysis of utterances, students using the Ad hoc technique were observed to spend the majority of their time in the lowest cognitive level of Bloom's taxonomy, Knowledge.

The ADR technique provides only a little more structure when compared to the Ad hoc technique. Students who conducted the ADR inspection needed to summarise and describe what each code segment (method) did. The technique is still general, in that it doesn't tell the inspector how to perform the inspection. Similar to the Ad hoc approach, students using ADR still have to work this out for themselves. ADR simply requires that method summaries be produced, in addition to a list of defects. Summarising and describing what task the code performs should fall into Bloom's Comprehension category.

Intuitively, it might be expected that ADR would have significantly more utterances in the Comprehension category when compared to other techniques. However, this was not seen to be the case. As with the Ad hoc technique, the vast majority of utterances fall into the Knowledge category. With very little structure given to the process, students operated mostly at Bloom's lowest cognition level, Knowledge. That is, for the majority of their time they were functioning in the Knowledge level, recalling relevant knowledge, such as how a control statement functions.

Examining the abstracts written, it may be that this method requires more time than Ad hoc. There were a total of 15 methods requiring an abstract, with no student completing all 15. One student completed ten and the remaining less than this. If more time was given for applying this method, then that may have changed the results.

The CBR technique is very structured; it describes what needs to be done and exactly how it should be implemented. Unlike the Ad hoc and ADR techniques, which leave much up to the inspector, the CBR method explicitly describes the process to be implemented. Yes/No questions are asked and depending on the answer determines if there is a possible defect or not. Results from this study show that inspectors using CBR were more likely to operate in the Bloom's Evaluation category, compared to inspectors utilising other less structured reading methods.

Students who used the CBR method also had a significantly lower number of utterances in the Knowledge category than those who carried out the ADR and Ad hoc inspections. A possible explanation for this could be that students understood they need only ask and answer the questions and their inspection is completed. Therefore they did not spend as much time in the Knowledge category, gaining a general, overall understanding of the class inspected, but rather completed the task at hand, answering the checklist questions to determine if a defect existed or not.

Figures 1, 2 and 3, and the statistical analysis suggest that student inspectors operate at higher cognitive levels when using more structured processes.

6. Recommendations

This study suggests, university level Computer Science and Software Engineering students should be taught reading techniques and critical thinking as they examine the artefacts of their discipline to develop higher cognition levels regarding the software they develop or maintain. In this study, the artefacts were source code programs, but in other situations it may include requirements documents, design documents or source code documentation.

The results reported in this study highlight the need to teach the usage of diverse techniques to students. For less experienced students, the beginning years of their degrees, should be introduced to techniques that have a well structured processes, with the aim of assisting them in operating at higher cognitive levels. This includes using CBR early in computer science and software engineering degree programs.

The CBR reading technique would form the base line reading methodology. It takes the inexperienced developer, with little or no programming background, giving them the experience of others, through the use of a checklist developed over time by experienced developers. Building upon this foundation, as the students become more experienced, further into their degree, less structured reading techniques, those that draw from their past knowledge and experiences as developers, would be introduced. Ideally this would start in the first year of a degree. The introduction of the CBR technique would give the student basics to think through when they start to examine their own code and that of others. As the degree progresses the more complicated reading techniques, relying on experience would be introduced allowing students to draw on their past knowledge and experiences.

7. Conclusions and future work

This paper builds the case for the need teach reading techniques, as an essential aspect of learning, throughout Computer Science and Software Engineering degrees. Without the skills of critical reading, thinking and analysis Computer Science and Software Engineering graduates lack a skill that they will be required to have from day one of their careers in industry. The continued research and application of reading techniques within Computer Science and Software Engineering is needed to better understand the ways in which these skills can be incorporated throughout whole degrees and not just as a simple lecture in one unit.

8. References

[1] Anderson, L.W., Krathwohl, D.R., Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, J. and Wittrock, M.C., A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Longman, New York, 2001.

[2] V.R Basili, Evolving and packaging reading technologies, Journal of Systems Software, 1997, 38(1), pp. 3-12.

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

[4] A. Dunsmore, Investigating effective inspection of object-oriented code, PhD thesis, Strathclyde University, U.K., 2002.

[5] A. Dunsmore, M. Roper, and M. Wood, Systematic object-oriented inspection - an empirical study, ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 135-44.

[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, 2003, 29(8), pp. 677-86.

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

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

[9] R.P. Gabriel, and R. Goldman, The erotic life of code, ACM Conference on Object-Oriented Programming, Systems and Languages, 2000. Available at: http://www.dreamsongs.com/MobSoftware.html, (Sep. 2008)

[10] T. Kelly, and J. Buckley, A context-aware analysis scheme for Bloom's Taxonomy, ICPC'06, Proceedings of 14th IEEE International Conference on Program Comprehension, 2006, pp. 275-284.

[11] O. Laitenberger, and J. DeBaud, An encompassing life cycle centric survey of software inspection, Journal of Systems and Software, 2000, 50(1), pp. 5-31.

[12] Moore, D.S., and McCabe G.P., Introduction to the Practice of Statistics, 4th ed., W.H. Freeman, 2002.

[13] F. Shull, I. Rus, and V. Basili, Improving software inspections by using reading techniques, ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 726-7.

[14] D.I.K. Sjoberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N. Liborg, and A.C. Rekdal, A Survey of Controlled Experiments in Software Engineering, Software Engineering, IEEE Transactions on Software Engineering, 2005, 31(9), pp. 733-53.

[15] Spinellis, D., Code Reading: The Open Source Perspective, Addison-Wesley Professional, 2003.

[16] E. Thompson, A. Luxton-Reilly, J.L. Whalley, M. Hu, and P. Robbins, Bloom's Taxonomy for CS Assessment 2008, Proceedings of the 10th Austrtalasian Computing Education Conference, 2008, pp. 155-61.