Measuring Cognition Levels in Collaborative Processes for Software Engineering Code Inspections

David A. McMeekin, Brian R. von Konsky, Elizabeth Chang and David JA Cooper

Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Enterprise Uni 4, De' Laeter Way, Technology Park, Bentley WA 6102, Australia
{d.mcmeekin,b.vonkonsky,e.chang,david.cooper}@curtin.edu.au

Abstract. This paper demonstrates that different software code inspection techniques have the potential to improve developer understanding of code being inspected to varying extents. This suggests that some code inspection techniques may be superior to others with respect to improving the efficacy of future inspections, harnessing collective wisdom, and extending team knowledge and networked intelligence. In particular, this paper reports results from a study of novice developers' cognitive development during a software inspection training exercise. We found that developers who performed a code inspection prior to modification tended to operate at higher cognitive levels beginning very early in the modification exercise. Those who had not performed an inspection tended to operate at lower cognitive levels for longer periods of time. Results highlight the importance of code inspections in increasing developers' understanding of a software system. We believe collaboration between academia and industry in studies such as these would benefit the three major stakeholders: academia, industry and graduates.

Keywords: Collaboration, Collective effort, Software inspections, Bloom's taxonomy, Programmer comprehension, Cognition development

1 Introduction

Industry-based software inspection processes are normally used to detect software defects. However, they can also have an impact on a developer's understanding of the system being inspected, with the potential to improve team cognition levels and the effectiveness of future collaborative inspection exercises.

A software inspection was used as a training exercise prior to developers adding functionality to code. During both the training and coding exercises, the software developers' cognitive levels were measured using the Context-Aware Analysis Schema [15]. One group of developers had not seen the code prior to adding functionality. The other group of developers had inspected the code immediately prior to adding the functionality using one of three inspection-reading techniques: Ad hoc reading, Abstraction Driven Reading or Checklist-Based Reading.

2 Industry Practice

Software Inspections are a practical methodology widely used in the ICT industry. They are a tried, tested, and effective method for the removal of defects from software early in the development life cycle [11]. Additionally, software inspections assist inspectors in developing greater insight and understanding into the artefact being inspected [16]. IDE's such as Netbeans, Eclipse, Xcode and Visual Studio, with their auto completion functions for example, has meant developers are warned of many potential errors before finishing writing the line of code. For example, when calling a function, the auto completion will display the method signature. This assists the developer to order the parameters correctly.

Reading software artefacts is an essential practice for producing high-quality software during a product's development and maintenance life cycle [2]. Inspection techniques/reading strategies are usually linked with verification and validation of software artefacts. Applying inspections to raise cognition levels and reading skills is an area that has not been well researched as an additional possible benefit of software inspections.

Traditionally software inspections are a collaborative task, typically comprising four inspectors. The first person is the moderator, who presides over and manages the team inspection process. The second is the designer responsible for the design of the code in question. The third is the implementer who translated the design document into code. The final participant is the tester who was responsible for writing and executing the test cases.

2.1 Practical Software Inspection Methods

Software inspections are implemented early in the development process to detect defects in the inspected artefact [11], and offer developers a structured method to examine software artefacts for defects [9]. Software inspections and their success in detecting defects is a well-researched area in software engineering [23].

Performing a code inspection prior to modifying the code has been shown to improve a developer's ability to carry out the required changes [19]. The inspection techniques tested in this study were: ad hoc, Abstraction-Driven Reading (ADR), and Checklist-Based Reading (CBR).

The ad hoc technique is understood to be the simplest inspection technique to use. No formal methodology is used when applying this method. The inspector is expected to thoroughly inspect the artefact using his/her personal experience as the guide [16].

This method's strength lies in giving the greatest freedom to the inspector as to how they execute the inspection [9]. Its greatest weakness, correspondingly, is uncovered by novice developers, who lack the necessary experience to effectively apply it [16].

The Abstraction-Driven Reading (ADR) technique was created in response to the delocalisation challenge Object-Orientation introduced to traditional inspections [8] [9]. The inspector reads code in a systematic way, writing natural language abstract specifications about each method and class. While reading each method, calls to

delocalised code are followed and the invoked code is also inspected. As the inspector systematically executes these tasks they also compile a list of detected defects.

A strength of this technique is the requirement for inspectors to develop reusable natural language descriptions of the inspected code. However, this comes with concomitant costs in time, and can be overwhelming for the inspector to attempt to grasp an understanding of the whole program.

Checklist-Based Reading inspections were formally introduced by Fagan [11] and are considered the standard inspection method used by software organisations today [16]. The inspector has a series of questions that guide their reading. The questions should be derived from historical data from within the organisation identifying defects detected in previous systems [12][13].

Each question on the list requires a yes or no answer. A yes answer implies no defect in the code at that location. A no answer indicates the possibility of a defect there and necessitates a closer examination of the code.

A strength of this technique is that the checklist is a product of prior inspections and captures organizational history with respect to the cause of prior defects. A weakness is that it is a highly structured process that can restrict inspectors from reading the code in a more natural manner.

3 Bloom's Taxonomy for Educational Objectives

3.1 Bloom's Cognitive Development

Bloom's taxonomy is a well-established categorisation of six different cognitive levels potentially demonstrated during learning [1] [4]. The categories range from the lowest to the highest level of cognitive learning. The classification is widely used in education systems throughout the world.

Each category, cited from [1], is listed and briefly described below, with an example of how each might be translated in a programmer's context:

Knowledge: "retrieving relevant knowledge from long-term memory." For the programmer this may be the specific recalling of an if-then-else statement.

Comprehension: "construct meaning from instructional messages, including oral, written, and graphic communication." For the programmer, summarising a method or code fragment.

Application: "carry out or use a procedure in the given situation." Demonstrated when the programmer makes a change in the code.

Analysis: "break material into constituent parts." Where the programmer describes a method or field's operation and role within the wider system.

Evaluation: "make judgements on criteria and standards." Here the programmer makes a judgement on the correctness or incorrectness of a part of the program.

Synthesis: "re-organise elements into a new pattern or structure." The programmer creates a new class, successfully integrating it into the wider system.

3.2 Industry-based Context-Aware Schema using Bloom's Taxonomy

Bloom's taxonomy has been used in many software engineering studies to examine developers' comprehension levels during different tasks [6][15][25][26][27]. Kelly and Buckley [15] developed a Context-Aware Schema for use with the taxonomy. The schema requires developers to "think-aloud" as they perform the different tasks required of them. Think-aloud is a process in which the participant verbalises thoughts and actions while carrying out the task [10].

The think-aloud data is recorded, transcribed and broken down into sentences or utterances. Each sentence or utterance is then categorised into a level within the taxonomy to identify the cognitive level at which the developer was operating. Each utterance is categorised upon both its content and the previous two utterances. This enables the utterance to be categorised within its applied context.

The original Context-Aware Schema [15] omitted the synthesis level, as their study was carried out in a maintenance environment. In our study we have introduced the synthesis category. This was because developers were required to add new functionality to the code. This new functionality was not fixing defects but rather extending the program to perform a new task.

4 Methodology for Collective Academic and Industry Learning

To investigate understanding arising as a result of the various code inspection strategies, a study was conducted in which novice developers were required to add new functionality to an existing software system. During this process their cognition levels were measured using the Context-Aware Schema [15].

The software system was the game of Battleship. It was a text-based implementation written in Java and contained seven classes in total. The Board.java class required new functionality; the ability to place ships in a diagonal down to the right manner. Participants were given 30 minutes to add this new functionality, and were required to think-aloud for the task's duration. When their time was up, participants stopped regardless of whether they had completed the task or not.

The study was advertised on the university campus and participants took part in their own time. No compensation was paid to participants and they were informed that participation had no influence whatsoever on their marks/grades in courses they were currently undertaking. Participants were required to be final year undergraduate studying Software Engineering, Computer Science or Information Technology.

Participants were provided with the following artefacts for the modification task:

- a natural language description of the system,
- a class diagram,
- the Board.java file (without defects),
- access to the other Java code in the system,
- a natural language modification request, and
- access to the Java APIs. All artefacts were online.

Prior to adding the functionality, four participant groups were established and individual members from three of the groups performed a 30-minute code inspection

on the Board.java class, searching for defects. The think-aloud data was also collected from the inspection task. Group One performed an ad hoc inspection, Group Two performed an ADR inspection, Group Three performed a CBR inspection, and Group Four did not perform an inspection. Participants performed two small training exercises prior to participating in the study. The first exercise involved using the assigned inspection technique. The second exercise detailed how to think-aloud.

Participants were informed that the defects seeded in the code were not syntax–related, as the code compiled and executed. They were searching for defects that would cause the system either to fail or produce incorrect output.

Research of this nature, based on empirical studies, is subject to internal and external validity threats. The first internal threat to this study was the selection threat. Selection threat is where participants are stacked to produce favourable results. To limit this threat, the study was advertised on campus, all final year students who asked to participate were admitted to the study, and all participants were randomly assigned to the different inspection technique groups.

The second internal threat, as with many software engineering studies, was variation in participant experience. In considering this threat, demographics were collected from participants in order to monitor discrepancies that may have arisen within the results from this.

The external validity threat in this study was the sample size. There is significant overhead involved when using the think-aloud method. The data must be collected, collated, transcribed, broken into utterances, and analysed. The sample size was kept to 20 as the research was attempting to identify any emerging trends within the data that could be pursued with larger data sets in the future. It must be noted that even with small sample sizes, although difficult to generalise to a larger body, significant differences may still be identified [20] warranting continued research.

5 Results on Effective Bloom's Cognition Development and Software Skill Training

Table 1 displays an utterance example from each of the 6 different categories of Bloom's taxonomy. The seventh category (Graph Number *0*) shown in Table 1 is Uncoded, and is not part of the taxonomy. Utterances in this category were either unintelligible or unrelated to the task at hand, such as talking on the mobile phone during the study.

Table 1. Example of utterances.

Graph Number	Bloom's level	Utterance Example
1	Knowledge	"while ship not sunk"
2	Comprehension	"this is a one to many relationship"
3	Application	"we need to cater for a new direction"
4	Analysis	"this is externally controlled"
5	Evaluation	"the call here is incorrect"
6	Creation	"creating a new method"
0	*Uncoded*	*"what food will we need for tonight"*

Figures 1, 2, 3 and 4 graph four different participants' utterances in order of their occurrence, categorised into the appropriate cognitive level using the Context-Aware Schema. The X-axis shows the order of the utterances and the Y-axis represents utterance's cognitive level.

Figures 5 and 6 are graphed in a similar manner, but the utterances are from the inspection each participant performed. Due to the large number of graphs, only samples of the participants have been displayed in this paper.

Figure 1 demonstrates almost 50% of participant 15's modification utterances were in the lowest cognitive level, Knowledge. The participant was unfamiliar with the code prior to receiving the modification request. Hence, in the 30 minutes given for the task, they needed to familiarise themselves with the code and then perform the modification. This low cognition level is indicative of the participant reading the code in an attempt to understand what task the code performed. Once this was sufficiently understood, the modification could then be attempted. After that point they began to operate at higher cognitive levels. This pattern was similarly repeated with all participants who did not perform a code inspection prior to modifying task.

Figure 2 graphs participant 5's modification utterances. It shows that participant 5 started with a small number of utterances in the low cognitive levels and then moved into the higher cognitive levels: synthesis, application and evaluation, remaining there for a large portion of the modification time.

Prior to performing the modification task, participant 5 had performed a CBR inspection. Figure 5 graphs participant 5's utterances from the CBR inspection. The graph shows participant 5 started the inspection with a small number of utterances in the lowest cognitive levels and then moved into the higher cognitive levels: Analysis and Evaluation.

Figure 3 shows participant 1 started with very few low-level utterances and then moved into the higher categories within the taxonomy. Before carrying out the code modification, participant 1 had performed an ad hoc inspection on the class. The participant also had a vast number of Uncoded utterances while performing the modification. The recording indicates this participant talked about unrelated things while performing the code changes.

In Figure 4, participant 17's utterances during the modification reflect a very similar story to those already described. Having performed, in this case, an ADR code inspection prior to the modification, the participant started with a small number of low-level cognitive utterances and then moved into the higher cognitive levels.

Figure 6 displays participant 1's utterances from an ad hoc code inspection. The ad hoc inspection technique is without structure and the inspector operates mostly in the lowest cognition levels. Comparing this to participant 15's modification utterances, Figure 1, both start at the very low cognitive levels. The two graphs appear very similar in the way their utterances are spread through the different categories, and yet one is from an inspection, Figure 6, and the other is from a modification, Figure 1. During the inspection, participant 1 is working to understand the code while participant 15 is working to add functionality to the code. As they are attempting to understand the code they operate at similar cognitive levels. One participant is looking for defects, the other looking to add functionality yet the cognitive levels are very similar.

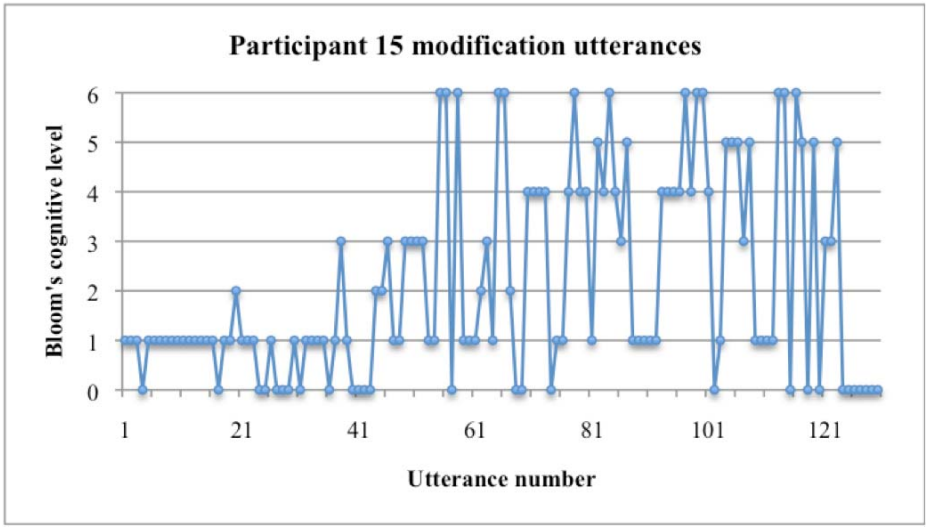

Fig. 1 Participant 15's modification utterances. Performed no inspection.

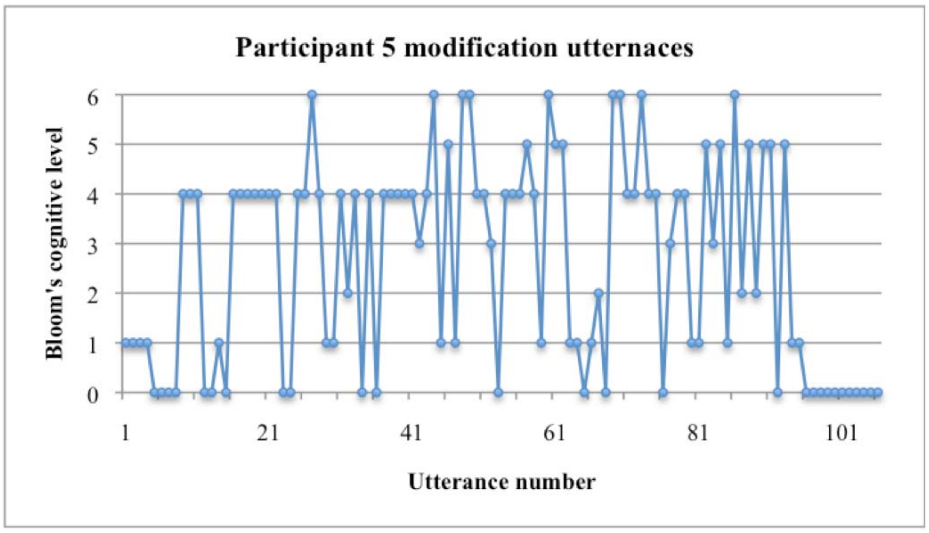

Fig. 2 Participant 5's modification utterances. Performed a CBR inspection.

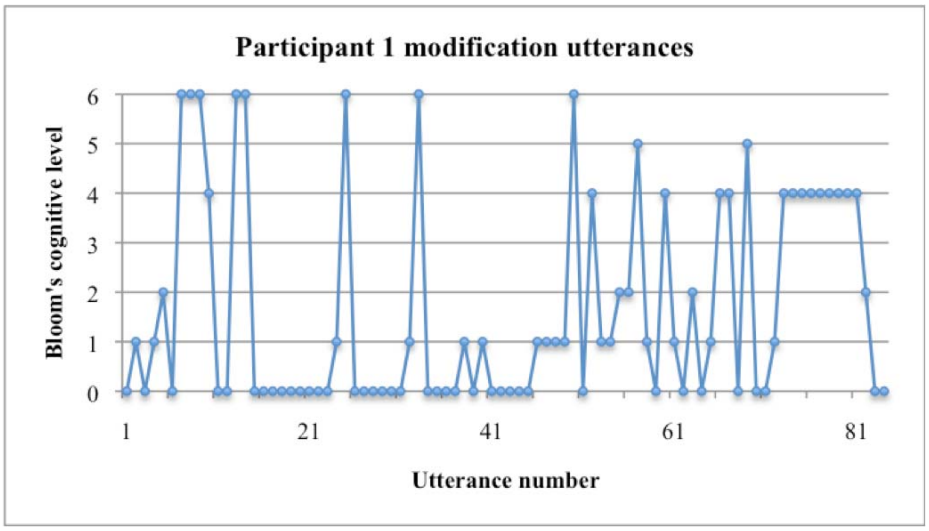

Fig. 3 Participant 1's modification utterances. Performed an ad hoc inspection.

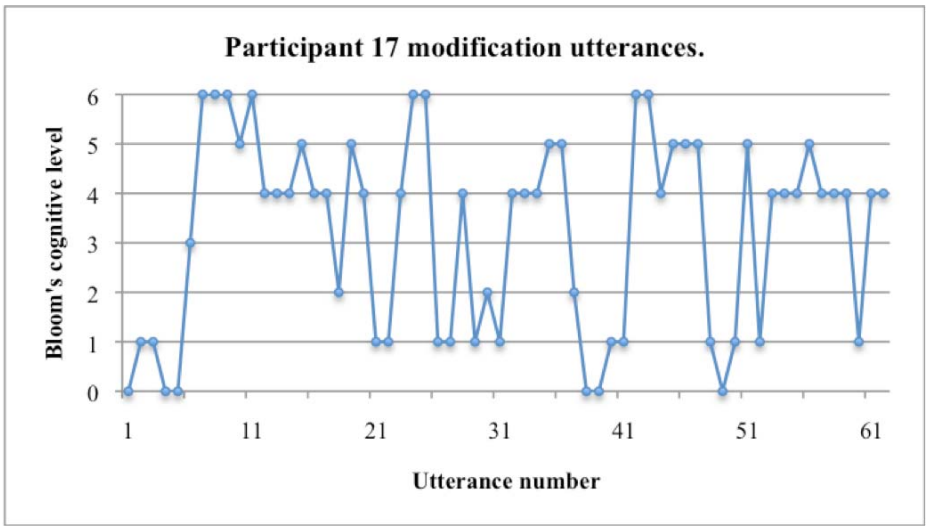

Fig. 4 Participant 17's modification utterances. Performed an ADR inspection.

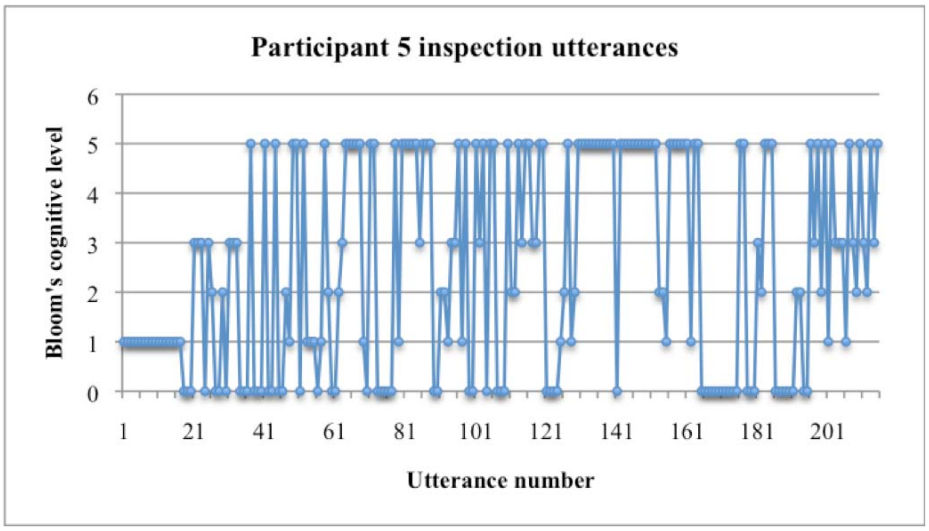

Fig. 5 Participant 5's utterances during inspection. Performed a CBR inspection.

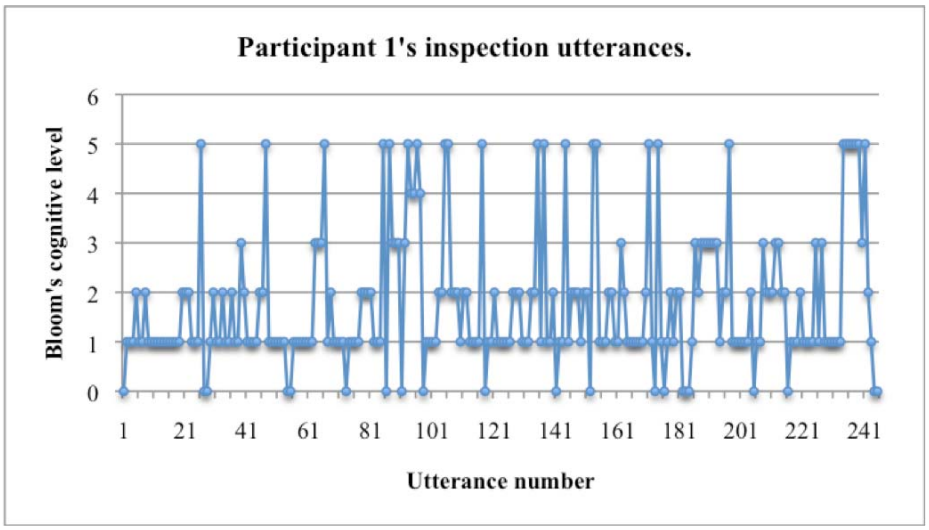

Fig. 6 Participant 1's inspection utterances. Performed an ad hoc inspection.

6 Discussion

These results demonstrate that performing a code inspection prior to adding functionality impacts novice developers' cognitive levels. Moreover, the various inspection techniques used affects developers cognitive levels differently.

Figure 6 shows that participant 1 consistently operated in the lower cognitive levels during the inspection. They were attempting to understand what the code does. When

comparing it to what the code should have done, they moved into the higher cognition levels.

Figure 1 shows participant 15, who did not perform an inspection prior to modifying the code, largely operated at the lower cognitive levels. This was similar to participant 1's utterances shown in Figure 6. Notably, both participants operated at similar cognitive levels yet were performing very different tasks: participant 1 was performing an ad hoc code inspection and participant 15 was attempting to add functionality. The ad hoc inspection technique, used by participant 1, is unstructured; the inspectors must use their own experience to successfully execute the inspection. The cognitive levels experienced when performing an ad hoc inspection are similar to those experienced when adding functionality to unfamiliar code. In the case of the ad hoc inspection, no direction is given and in the case of the functionality being added, the code is unknown and must first be understood in order for it to be modified. However, when participant 1 moved into the modification task, they operated at the higher cognitive levels, application and synthesis, from very early unlike participant 15 when they were performing the same task (as noted earlier, participant 1 also talked about unrelated things while performing the tasks).

Participant 5, shown in Figure 5, commenced their code inspection with a small number of utterances in the low cognitive levels and then moved into the higher levels. At the higher cognitive levels, as with participant one, they were judging the code for correctness. However, participant five operated more consistently and for a longer time period at these higher cognitive levels than participant 1 did.

Participants using the CBR inspection technique operated at higher cognitive levels and when they moved to add new functionality they continued to operate at the higher cognitive levels: application, synthesis and evaluation more consistently and for longer time periods. The CBR inspection technique facilitated higher cognitive levels within the inspection and the inspector, when making modifications, appeared to continue to function at these higher levels.

We found that in our study of novice developers' cognitive development, during the practical-based skill training exercise, the developers who performed the inspection prior to modification tended to operate at higher cognitive levels from very early on while those who had not performed the inspection tended to operate at lower cognitive levels for longer periods of time.

The results highlight the important role software inspections can play in increasing developer comprehension of a system. They also support the notion that when introduced to a new program or code, one must first go through an initial stage of low cognition levels to gain a basic fundamental understanding of the code, its operands and operations. For the novice developer, the less structured the process for working through this stage, the longer they operate at the lower cognitive levels of the taxonomy. Conversely, the more structured the technique used to familiarise themselves with the code they are working with, the faster they move into the higher cognitive levels of the taxonomy.

The CBR inspection structure facilitates inspectors to function within the highest cognitive levels, above that of both the ad hoc and ADR. This is due to the question and answer nature of the checklist, which requires the inspector to judge the code for correctness.

As software systems continue to simplify the user experience while increasing the complexity for developers, it is important that effective methods are developed to assist novice developers joining these teams to understand the code-bases they are working on as quickly as possible. This will service quality education with skilled graduates that meet the ICT industry needs.

7 Conclusions

This study, and ultimately the three major stakeholders, academia, industry and students, could all benefit from collaborating on future research investigating the benefits of inspections to improve collaborative design cognition and extending team knowledge.

Industry's benefit from collaboration with academia would be in assisting to produce higher quality graduate developers. These graduates' skills would have already been tried and tested in the environment of with industry-based code. This could aid in reducing costs related to graduate training and also reduce the amount of productive time senior developers lose when answering rudimentary questions from new developers.

Collaboration between academia and industry would result in students also benefitting. Prior to moving into the work force, students will have seen and worked on industry based code. Novice developers' exposure to this type of code would create an awareness of the complexities of the code they will be working with when they move from academia into industry. Their education would have covered both the theoretical side and the industry side of issues faced by developers.

The call must be made for increased collaboration between industry and academia. The use of ICT continues to become more and more ubiquitous and the underlying complexities of ICT continue to increase. Collaborative research between academia and industry into effective reading strategies to improve developer comprehension is essential in raising the quality of software being produced by increasing the quality of software development graduates.

The disconnect between academia, their ICT graduates and ICT industries is as common as the gap between business objectives and IT solutions. Despite the strong shortage of ICT skilled professionals in all industries, academia has a hard time creating ICT graduates that meet industry needs. Currently, existing ICT education and the rest of the ICT industry throughout the world are out of sync. For example, the evolution cycle of Technology is 6 months, but in academia, most curriculums change approximately every 3-5 years. Without a collective academic industry learning effort, students will study outdated technology and practices that will be even more outdated in 3 years time, when they graduate. Therefore, the collective effort will help to keep pace between ICT revolutions and state-of-the-art education, enabling global knowledge, networked intelligence, and extended knowledge to penetrate the educational sector.

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. Basili, V.R., Evolving and packaging reading technologies, Journal of Systems Software, 38(1), pp. 3-12, (1997).
3. Bergantz, D., and Hassell, J., Information relationships in prolog programs: how do programmers comprehend functionality? Int. J. Man-Mach. Stud., 35(3):313–328, (1991).
4. Bloom, B., Taxonomy of Educational Objectives Cognitive Domain, David McKay Company, Inc., (1956).
5. Brooks. R., Towards a theory of the comprehension of computer programs. International Journal of Man–Machine Studies, 18(6), pp. 543–554, (1983).
6. Cooper, D., von Konsky, B., Robey, M., and McMeekin, D.A., Obstacles to comprehension in usage based reading, Proc. 18th Australian Conference on Software Engineering (ASWEC 07) pages 233–244, (2007).
7. Dunsmore, A., Investigating effective inspection of object-oriented code, PhD thesis, Strathclyde University, U.K., (2002).
8. Dunsmore, A., Roper, M., and Wood, M., Systematic object-oriented inspection - an empirical study, ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, pp. 135-44, (2001).
9. Dunsmore, A., Roper, M., and Wood, M., The development and evaluation of three diverse techniques for object-orientated code inspection, IEEE Transactions on Software Engineering, 29(8), pp. 677-86, (2003).
10. Ericsson, K.A. and Simon, H.A., Protocol Analysis, The MIT Press, 1993.
11. Fagan, M.E., Design and code inspections to reduce errors in program development, IBM Systems Journal, 15(3), pp. 182-211, (1976).
12. Fisher, C., Advancing the study of programming with computer-aided protocol analysis. pages 198–216, (1987).
13. Gilb, T., and Graham, D., Software Inspection. Addison–Wesley, Wokingham, 1993.
14. Humphrey, W., A Discipline for Software Engineering. Addison–Wesley, Boston, 1995.
15. Kelly, T., and Buckley, J., A context-aware analysis scheme for Bloom's Taxonomy, ICPC'06, Proceedings of 14th IEEE International Conference on Program Comprehension, pp. 275-284, (2006).
16. Laitenberger, O., and DeBaud, J., An encompassing life cycle centric survey of software inspection, Journal of Systems and Software, 50(1), pp. 5-31, (2000).
17. Littman, D., Pinto, J., Letovsky, S., and Soloway, E., Mental models and software maintenance. Journal of Systems Software, 7(4) pp. 341–355, (1987).
18. von Mayrhauser A., and Vans, A.,. Identification of dynamic comprehension processes during large scale maintenance. Transactions on Software Engineering, 22(6) pp. 424–437, (1996).
19. McMeekin, D.A., B. R. von Konsky, E. Chang and D.J.A. Cooper, Checklist Based Reading's Influence on a Developer's Understanding, Proc. 19th Australian Conference on Software Engineering (ASWEC 08), pp. 489-496, (2008).
20. Moore, D.S., and McCabe G.P., Introduction to the Practice of Statistics, 4th ed., W.H. Freeman, (2002).
21. Shull, F., Rus, I., and Basili, V., Improving software inspections by using reading techniques, ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, pp. 726-7, (2001).

22. Siy, H. and Votta, L. Does the modern code inspection have value? In Software Maintenance. Proceedings. IEEE International Conference on, pages 281–289, (2001).
23. Sjoberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N., and Rekdal, A.C., A Survey of Controlled Experiments in Software Engineering, Software Engineering, IEEE Transactions on Software Engineering, 31(9), pp. 733-53, (2005).
24. Tyran, C. K., and George, J. F., Improving software inspections with group process support. Communications of the ACM, 45(9) pp. 87–92, (2002).
25. Xu, S., and Rajlich, V., Cognitive process during program debugging. Proc. Third IEEE International Conference on Cognitive Informatics, pp. 176–182, 2004.
26. Xu, S., and Rajlich, V., Dialog-based protocol: an empirical research method for cognitive activities in software engineering Proc. International Symposium on Empirical Software Engineering, (2005).
27. Xu, S., and Rajlich, V., and Marcus, A., An empirical study of programmer learning during incremental software development. Proc. Fourth IEEE Conference on Cognitive Informatics (ICCI 2005), pages 340–349, 2005.