Checklist Based Reading's Influence on a Developer's Understanding

David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper
Curtin University of Technology
Department of Computing,
Digital Ecosystems and Business Intelligence Institute
Perth, Western Australia
d.mcmeekin@curtin.edu.au

Abstract

This paper addresses the influence the Checklist Based Reading inspection technique has on a developer's ability to modify inspected code. Traditionally, inspections have been used to detect defects within the development life cycle. This research identified a correlation between the number of defects detected and the successful code extensions for new functionality unrelated to the defects. Participants reported that having completed a checklist inspection, modifying the code was easier because the inspection had given them an understanding of the code that would not have existed otherwise. The results also showed a significant difference in how developers systematically modified code after completing a checklist inspection when compared to those who had not performed a checklist inspection. This study has shown that applying software inspections for purposes other than defect detection include software understanding and comprehension.

1 Introduction

Traditionally, software inspections have been implemented to detect defects. A vast amount of empirical research has been conducted using inspections and results have shown over 80% of defects can be removed prior to release, as well as saving large amounts of programmer time [5, 6]. The dominant inspection technique remains the Checklist Based Reading technique (CBR) [9, 16].

Software development methodologies, tools and the technology used to develop and deploy software have evolved since inspections were introduced. Siy and Votta [12] indicate with these evolutions in areas such as strongly typed languages, automated tools and improved testing environments, many defect groupings no longer exist. These changes can cause some managers, already questioning the continued relevance of inspections or their implementation, to look at redirecting resources away from inspections into other areas. This research investigates the benefits of using software inspections to increase a developer's understanding of the code for the purposes of adding new functionality to existing software systems.

This paper reports on an empirical study conducted to:

- examine if participants thought an inspection assisted them in performing subsequent modifications (adding functionality) to the inspected code
- investigate if there was a relationship between the number of defects a participant discovered and the number of modifications they successfully made (modifications were not related to the defects)
- determine if participants approach code modifications in significantly different ways if they have previously performed a code inspection on that code

2 Background and Definitions

Software inspections are a process implemented in the software development life-cycle to detect defects in the software artefacts under inspection [5, 15, 11]. Laitenberger and DeBaud [9] state a purpose for software inspections is for inspectors to gain insightful understandings of the artefact being inspected. Sjøberg *et al.* [13] describe a large body of literature on topics in software inspections. This literature generally reports on inspection techniques, the way in which inspection techniques were implemented, and the resulting number of defects discovered. Siy and Votta [12] examine the usefulness of inspections in non-traditional areas: maintainability, code understandability, reduction in redundant code, increased safety, improved portability and improved documentation quality.

2.1 Inspection Techniques

The Ad-hoc technique is an informal but common inspection technique [9]. Applying this technique requires no formal training or instruction, as there are no instructions or directions for the inspector. The underlying assumption is that the inspector will carry out a thorough and systematic inspection of the artefact by using their personal experience and understanding [9]. This method is considered very effective, but is best applied by an experienced developer. Dunsmore [4] points out that a new developer may not have the experience and understanding needed to successfully apply this inspection technique.

The CBR technique, first described by Fagan [5] is the most common inspection methodology used today [16]. In CBR, the inspector answers a series of questions regarding the artefact being inspected. A 'yes' answer indicates there is no defect and a 'no' answer indicates the possibility of a defect requiring further investigation. The checklist questions must be created using historical defect data from within the organization [7, 8]. The checklist should also fit on one side of a single sheet of paper [1], preventing an inspector from having to continuously turn pages. CBR is a highly structured inspection methodology with little space for the inspector to apply previously gained expertise. This technique is beneficial for new developers as it gives clear instructions and structure in how to go about the inspection and to identify defects. Yet for the experienced developer, this methodology has been reported as being limiting, tending to focus developers on program structure rather than business logic (work by the authors pending publication).

CBR is considered the industry standard inspection technique and Thelin *et al.* [14] recommended it be used as the baseline inspection technique in empirical studies. Consequently, CBR was chosen to be tested first in this study.

3 Methodology

A pilot study was carried out in which participants performed a CBR code inspection. After completing the code inspection, participants were informed of modifications to be implemented within the code. This study investigated the objectives presented in Section 1 and was approved by the Human Research Ethics Committee at Curtin University of Technology.

3.1 Artefacts

The software system inspected was created specifically for use in a tutorial in a third year software engineering course. It was a navigation recording system in which users entered their current latitudinal and longitudinal positions. The class inspected stored these positions, gave access to past positions and enabled the distance between two recorded points to be calculated. The modification was to store the user's altitudinal position and use it in the calculations. The class contained 126 executable lines of code.

The artefacts for the inspection were paper based and participants were requested not to compile or execute the code. Participants were informed that the code compiled, executed and produced results. During the inspection participants also had access to the Java API documentation.

Inspectors were presented the following artefacts:

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

3.2 Choosing Participants

Two participant groupings were established. Group one consisted of students in the third year of their undergraduate bachelor degrees in Computer Science, Information Technology or Software Engineering. Participants were required to have passed two introductory Java programming courses and two software engineering courses. These criteria were established in order for an assumption to be made that participants in group one had similar level base knowledge. Group two consisted of students doing the undergraduate bachelor degrees in Software Engineering. These students met the same requirements as group one.

Group one performed the code inspection followed by the modification. Group two performed only the code modification and were given the natural language specification, class diagram of the system and questionnaire. Group two participants were recruited for the purpose of determining if participants approached code modifications differently if they had previously performed a code inspection on that code.

3.3 Carrying out the Experimental Study

The first section of this study extended a tutorial used in a third year software engineering course. The tutorial required participants to perform a CBR inspection on a single class. The second section of this study required participants to modify the code they had just inspected. Participation within this section was voluntary and was not part of the course and no part of the material or learning done through the study was examinable within participants' degree program course work. Upon completing the inspection tutorial, students were presented with the option to participate in this study. The study was conducted immediately following the

inspection in the participant's own time. No payments were made to participants.

The inspection process was an individual task and participants were required not to interact with others during the inspection. Participants were given 30 minutes to perform the inspection. During the tutorial's first 10 minutes, a small inspection training example was conducted demonstrating how to conduct the inspection. Using the checklist provided, participants inspected the code by answering each question with a "yes" or "no." Participants were advised to read the natural language specification first, followed by the class diagram. Each defect discovered was recorded in a defect recording sheet with the line number it appeared on and a sentence or two describing it. Participants did not have to correct the defect, just identify it.

The study's second section required participants to add new functionality to the code inspected. The new functionality requirement was explained to participants and they were given 30 minutes to make the modifications. The modification was an individual task and participants were not permitted to interact with others during the modification. The modification was conducted online and participants were able to compile their code as often as they chose.

3.4 Seeded Defects

Twelve defects were seeded in the inspected class. An example was three defects that were replications of the same error type, i.e. when writing code. It is not unusual to copy and paste similar code and then makes changes to that code. For this defect, the original code contained a logic error in the processing of an if-then-else statement prior to copying and pasting and hence was replicated in three different locations. The remaining defects were seeded based upon prior research [2, 3, 4] and also errors the authors were known to make while writing code.

3.5 The Modification

The additional functionality to be implemented was not correcting all defects previously detected. The new functionality was to allow the class to store the user's altitude. Participants were requested to add this functionality ensuring that their changes worked appropriately and produced correct results. The defects remained in the code and hence those defects that affected the modification needed to be fixed to ensure the added functionality returned correct results. The model solution required 37 modifications. Participants from both groups were given 30 minutes to perform this modification. During this time a screenshot was automatically taken every five seconds capturing:

- the code being viewed;
- the changes to the code;
- the order in which changes were made; and
- the way in which they went about making those changes.

3.6 Threats to Validity

Empirical research is subject to internal and external validity threats. In this study, several internal threats were identified. The selection threat was the first internal threat. This is where the participant selection can be stacked in order to produce more favourable results. In order to limit this effect, a general invitation was issued to students to participate within the study. Students were not approached individually and students who asked to participate did participate. No student was refused participation.

The second internal threat to this study, and to many software engineering studies, is that of a participant's experience. It is possible that some participants will have had a small amount of industry experience, some may have had considerable industry experience and there may be some who have had no industry experience. This is further complicated, as some may have industry experience but nothing reflecting what is required of them in this study. In order to reduce or monitor this threat, demographic data was collected from each participant to note if these situations were present within this participant grouping. Participants within each group were of similar experience levels and met the minimum requirements described in Section 3.2.

An external validity threat to this study was in the sample population. Students who participated may not be representative of the wider community. In order to minimise this effect, students were invited to participate and participation was voluntary. Group two participants, as noted, were completing a degree in Software Engineering. These students were in their fourth year and were participating in an industry based project. This experience was on a part time basis and still within the context of being part of a predefined undergraduate course. To minimise the threat to validity, these students were in group two and their data was only used to compare how participants implemented modifications when they had and had not previously carried out a code inspection.

Another external threat was in defect seeding was artificial and hence may not represent defect types currently encountered in the software industry. Section 3.4 addresses the manner in which defects were seeded.

4 Data Collection, Analysis and Results

During this study, the defect data was collected by participants entering data into a defect recording sheet. The

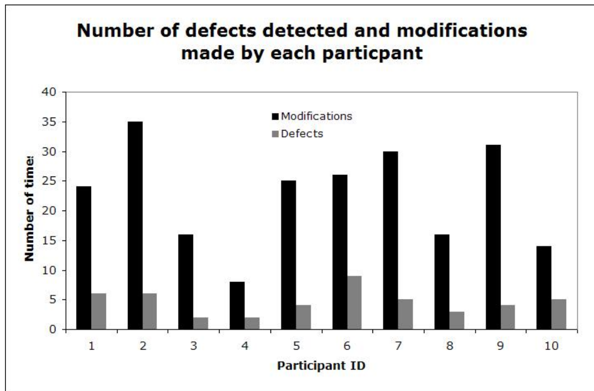

Figure 1. Number of defects found and modifications made by each participant.

false positive data collected was generated through examining the defect reporting data. The modification data was collected via the automatic screen-shots and the qualitative data was collected via a demographic sheet and a questionnaire participants completed. The quantitative data was analysed using the R (R 2.5.0 GUI 1.19 (4308)) statistical software package.

Eighteen participants took part in this study. Group one contained 11 participants and group two contained seven. There was one attrition in group one's data set, participant number three's results were removed from the data as they failed to detect a single defect and only reported modifications. In the screen shot analysis, four participants' results from group one were excluded from the set due to a server error that prevented a large portion of the data from being written to disk.

4.1 CBR Inspection followed by modification

Figure 1 shows the number of defects detected and the number of modifications made by participants in group one. In viewing this graph, and other remaining comparisons between these groupings, it must be noted that there were 12 seeded defects and 37 modifications. Figure 2 and Figure 3 are the box plots from group one's defect detection and modification data. The plots show there are no outliers and results are normally distributed.

Table 1 lists the descriptive statistics for the defects detected and modifications made. The mean number of defects detected is similar to those found in a previous study conducted by the authors where inspections were a new process that student participants were being introduced to [10].

Figure 4 demonstrates the relationship between the number of defects detected within the 30 minute inspection and

	Defects	Modifications
Mean	4.60	22.50
Std. Deviation	2.12	8.64
Std. Error	0.67	2.73
Minimum	2	8
Maximum	9	35

Table 1. Descriptive statistics from group 1.

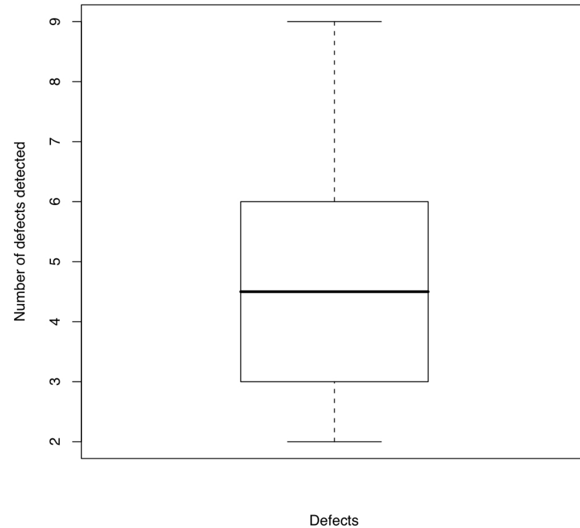

Figure 2. Boxplot of the defects detected by participant group one.

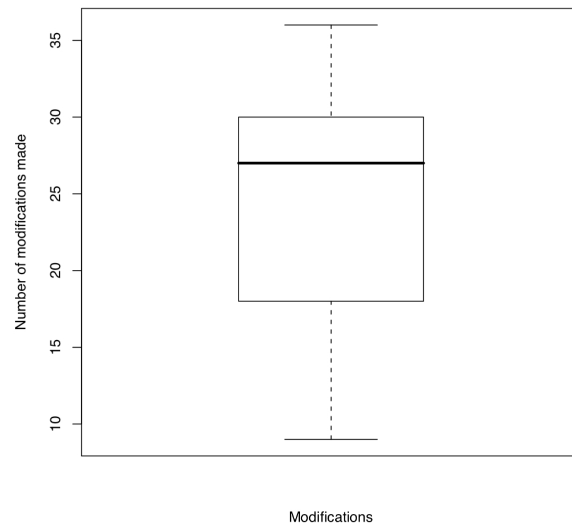

Figure 3. Boxplot of the modifications made by participant group one.

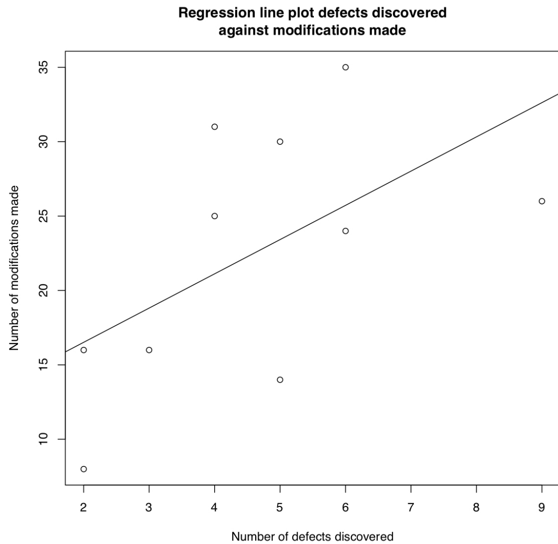

Figure 4. Scatterplot demonstrating the relationship between discovering defects and modification ability.

the number of modifications each participant made during the 30 minute modification session. A linear regression analysis was performed on the results, R-squared = 0.32 and the line of best fit is shown on the graph. The line and R-squared values indicate a weak correlation between the number of defects detected and the number of successful modifications made.

A Pearson Product-Moment correlation analysis was also performed upon the data, r = 0.56 and p-value = 0.09. These two values indicate a significant correlation, between the number of defects discovered and the number of successful modifications made to the code, at the 90% confidence interval.

4.2 Defect and Modification Types

Figure 5 displays how many times each defect was detected. Defect 1, 5, 9 and 11 had the highest detection levels. The code for defects 9, 10 and 11 assign an incorrect Boolean value when validating latitude, longitude and map number respectively. Defect 1 has incorrect parameter ordering, longitude and latitude were reversed. Defect 5 was the failure to use a parameter within the method. When using the checklist and walking through the code these defects were expected to be detected. The basis for this expectation was a checklist question which directly asked about this defect type. Defects 6 and 7 were not detected. Defect 6 used the '==' to compare doubles for equality and defect 7 used '==' to compare Vector objects for value equality.

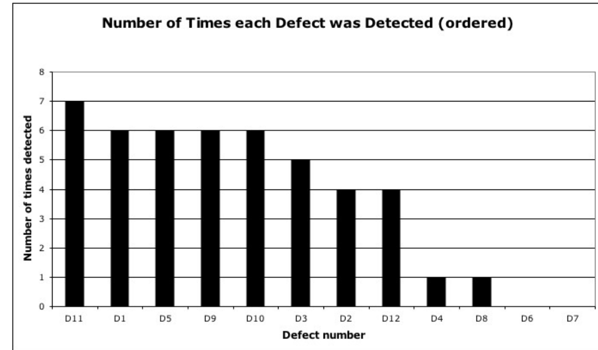

Figure 5. Number of times each defect was found (ordered from highest to lowest).

The checklist did not ask a direct question related to this defect type. From examining the data, detection of these defect types would have required a question directly asking about this error type. A checklist guides inspectors as they search for defects, but cannot be expected to ask every possible question as this would lead to the list length violating the recommendation that it be no more than 1 page long [1]. Defect 4 was detected once. This defect returns a Vector object, but in Java it actually returns a reference to the Vector object. This may cause the system to fail in an unexpected manner. Data expected to be in the Vector is no longer there or has been modified as the operations performed on the Vector object were in fact performed on the original object.

Figure 6 displays the number of times each modification was made. Modifications 1, 2 and 3 were made by all participants. These were adding the class field altitude and the class constants for the maximum and minimum values for it. The first modification was essential as without it no other modifications were needed. Modifications 2 and 3 were not always the second and third modification performed but on several occasions they were implemented when participants realised they were needed. Modifications 5, 8 and 21 were completed the least number of times. Modifications 5 and 8 were in two different constructor calls both requiring the initial Vector size be modified for it to grow in the correct increment sizes if needed. Modification 21 required a tolerance value be created for accurate comparison between doubles. Although this was not directly related to the defect of the same nature, it was not surprising that this was not completed as a defect of the same nature was also not detected. These two instances highlight the need for checklists to be continuously evolving as the defect that was not detected has now been introduce in a third location inside the inspected class.

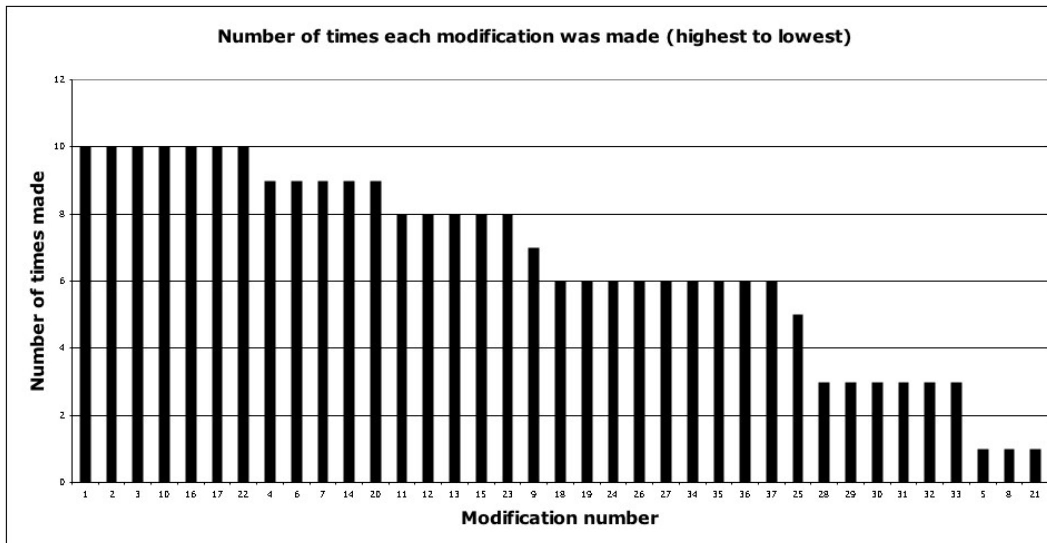

Figure 6. Number of times each modification was made (ordered).

4.3 Analysis of Participants' Perceptions

Upon completing the modification, participants were asked a series of questions regarding the inspection and modification. These questions provided qualitative data related to the impact participants thought the inspection had upon their ability to modify the code. Three questions were:

1. Did inspecting the code prior to modifying it assist you to make the changes: Yes or No?
2. From doing the inspection and changes, would you find it easier to add a new class: Yes of No? and
3. Do you agree with this statement: Performing the code inspection helped me to better understand the software: Strongly agree, Agree, Disagree, Strongly disagree

To question one, all participants answered yes. Participant three, who reported no detected defects also answered yes to this question. Even though participant three found no defects in the inspection process, in their opinion, it is still better prepared him to modify the code than having performed no inspection. This indicates that code inspections may be successful at improving the inspector's understanding, even though it was not entirely successful at detecting defects. For the second question, 9 participants answered yes and one stated no. When justifying their response to the second question, participants stated things such as: 'Easy to see the public methods for purposes for interfacing with a class,' 'seen how public methods work,' 'gained better understanding,' 'comfortable to work with the code,' 'because you have looked at the code and understood it.' These statements indicate the participants thought they had a better grasp and understanding of the code to be modified and added to from the inspection because of the knowledge they acquired about the code during the inspection. With the third question, 7 strongly agreed, 2 agreed and 1 failed to answer the question. This qualitative data clearly indicates, in the participants' minds, performing an inspection prior to modifying the code assisted their understanding and comprehension of the system enabling them to add the functionality and, had they been asked, to add other functionality to the system through adding more classes.

4.4 The Modification Order

The screen shots taken as participants modified the code showed the manner in which participants went about performing the modifications. Two sample solutions were created by the authors demonstrating two possible ways the modifications could be made. The base assumption used in both solutions was: to perform any modification to the code the class field must be present. Following this, the constants that controlled the class field's maximum and minimum values needed to be established. The next step was to initialise the class field in the constructors. At this stage the two sample solutions begin to vary. The first solution then continued to work through the class in a top to bottom manner. The second solution worked through the code by modifying it on an as encountered make the change manner. For example, if the participant was modifying method A and that method called method B, the participant would immediately move to method B, implement the needed changes there and then return to method A and continue modifying it. If while modifying method B, it called method C the same pattern as described would be followed. This continued until all

Group	Sample Solution 1	Sample Solution 2
One	0.81	0.51
Two	0.37	0.21

Table 2. The average R^2 values for the way in which each group modified the code.

modifications were completed.

Table 2 shows the average R^2 values for the way in which each group went about modifying the code. A two way t-test was carried out comparing how the two groups implemented the modifications compared to sample solution one, top to bottom of the class. A p-value of 0.001 was returned. This shows a significant difference in how the code was modified between the two groups. The participants who performed the inspection modified the code in a more systematic manner, one that more closely reflected sample solution one, than those participants who did not perform the inspection.

A two way t-test was carried out comparing how the two groups implemented the modifications compared to sample solution two, starting at the top and modifying the code on a as encountered and/or used method. A p-value of 0.02 was returned. This shows a significant difference in how the code was modified between the two groups. The participants who performed the inspection modified the code in a more systematic manner, one that more closely reflected sample solution two, than those participants who did not perform the inspection.

These results indicate that having performed an inspection, developers were able to modify the code by systematically working through it in a structured manner. Those who did not perform the inspection worked through the code but in an ad-hoc manner. This may be attributed to the fact that the developers knew the code structure as well as the location of methods that needed to be modified. With this knowledge they were able to move more directly to the code areas needing to be changed than those who had not performed the inspection and therefore also needed to search the code looking for where the changes may need to be made. The results also show that among participants who had performed the code inspection, when modifying code they tended to use a solution resembling sample solution one, working from top to bottom of the class.

5 Research Findings

This pilot study investigated if participants thought an inspection assisted them in performing modifications, investigated if a relationship existed between the number of defects detected and the number of successful modifications the inspector made to the code while and observing the way in which participants performed code modifications. The results clearly indicate that inspectors strongly believe performing an inspection prior to modifying code helps their understanding of the code and their ability to carry out the modification. The inspectors also believe that performing the inspection would assist them to add more functionality to the system due to their acquired knowledge about the system stemming directly from the inspection performed.

The statistical analysis results indicated there is a significant influence at the 90% confidence interval between the number of defects detected by an inspector and the number of successful modifications they are able to make (n = 18).

By performing an inspection prior to modifying the code, developers more systematically, succinctly and directly applied the needed modifications to the code. Further investigation of this point is needed to measure time savings that may be introduced in code modification by having developers first inspect the code and then perform enhancements and modifications to it.

6 Conclusion

This study's results clearly demonstrate that performing a CBR inspection positively affects a developer's ability to modify code. After performing a CBR inspection developers worked changes through the code more systematically than those who did not perform a CBR inspection. Participants ranked their system understanding and ability to modify the code as higher than if they had not performed the inspection. The application of inspections into the area of program comprehension is an under researched field. The varying inspection techniques need to be applied and tested in different development environments to measure how they affect developers' program comprehension in differing development environments and improve overall software quality.

References

[1] B. Brykczynski. A survey of software inspection checklists. *SIGSOFT Softw. Eng. Notes*, 24(1):82, 1999.

[2] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 467–476, Limerick, Ireland, 2000.

[3] A. Dunsmore, M. Roper, and M. Wood. Systematic object-oriented inspection - an empirical study. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 135–144, Toronto, Ontario, Canada, 2001.

[4] A. Dunsmore, M. Roper, and M. Wood. Further investigations into the development and evaluation of reading techniques for object-oriented code inspection. In *ICSE '02:*

Proceedings of the 24th International Conference on Software Engineering, pages 135–144, Orlando, Florida, U.S.A, 2002.

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

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

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

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

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

[10] D. A. McMeekin. A comparison of code inspection techniques. Honours Thesis, Curtin University of Technology, 2005.

[11] 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.

[12] H. Siy and L. Votta. Does the modern code inspection have value? *icsm*, 00:281, 2001.

[13] D. Sjøberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasavanovic, N. Liborg, and A. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, Sept. 2005.

[14] 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.

[15] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–56, Denver, Colorado, United States, 1999.

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