Digital Ecosystems and Business Intelligence Institute

A Software Inspection Methodology for Cognitive Improvement in Software Engineering

David Andrew McMeekin

This thesis is presented for the Degree of

Doctor of Philosophy

of

Curtin University of Technology

February 2010

Declaration

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgement has been made.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

Signature: ……

Date: ……

In loving memory of my sister:

Jo-ann Marie McMeekin 7th March 1967 – 22nd January 2007

Life was just too short for you!

Your passing inspires me to live with no regrets and remember

"Keep the important important and everything else is simply… well not."

I miss you everyday.

Abstract

This thesis examines software inspections application in a non-traditional use through examining the cognitive levels developers demonstrate while carrying out software inspection tasks. These levels are examined in order to assist in increasing developers' ability to understand, maintain and evolve software systems.

The results from several empirical studies carried out are presented. These indicate several important findings: student software developers find structured reading techniques more helpful as an aid than less structured reading techniques, while professional developers find the more structured techniques do not allow their experience to be applied to the problem at hand; there is a correlation between the effectiveness of a software inspection and an inspector's ability to successfully add new functionality to the inspected software artefact; the cognitive levels that student developers functioned at while carrying out software inspection tasks were at higher orders of thinking when structured inspection techniques were implemented than when unstructured techniques were applied.

From the empirical results a mapping has been created of several software inspection techniques to the cognitive process models they support and the cognitive levels, as measured using Bloom's Taxonomy that they facilitate. This mapping is to understand the impact carrying out a software inspection has upon a developer's cognitive understanding of the inspected system.

The knowledge and understanding of the findings of this research has culminated in the creation of a code reading methodology to increase the cognitive level software developers operate at while reading software code. The reading methodology distinguishes where in undergraduate and software developer training courses different software inspection reading techniques are to be implemented in order to maximise a software developer's code reading ability dependent upon their experience level.

Acknowledgements

There are many people who have supported me throughout my PhD that I am truly thankful to. I would like to thank Dr Andrew Marriott who was the first person to challenge me to think beyond Undergraduate study. Dr Michael Robey, who started out as one of my supervisors and has been an encouragement throughout. To my supervisor Dr Brian von Konsky, thank you, I certainly don't think I am the easiest person to supervise, but you have put up with all that I could throw at you throughout these 4 years, Brian thank you simply doesn't seem enough. To Professor Elizabeth Chang, thankyou for adopting me into the Digital Ecosystems Business Intelligence Institute (DEBII) and taking me on as one of your PhD students. Your endless support and encouragement has meant more to me than you could possible imagine. To the team of researchers and fellow PhD students at DEBII, it was fun, thanks.

My family: to my parents, thank you!! To my three children Josiah David, Marta Katarina and Jozua Andrew, you have endured more through this PhD than kids should endure, it wasn't even your degree. You three wonderful kids light up my life everyday; I love you more than you will ever know. To the most wonderful woman I have ever encountered, my darling wife Katharina, words cannot be said that will convey to you the gratitude I have for you and all that you have put up with throughout these 4 years. I am truly grateful for your endurance and your never-ending source of encouragement to me: thanks, I love you.

Finally, thanks be to God, who without, none of this would have been possible. I am truly humbled by Your goodness to me and my family, thank you!

Publications Related to this Thesis

Refereed International Conference Papers:

- David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper, *Mapping a Sequence Diagram to the Related Code: Cognitive Levels Expressed by Developers*, 7th IEEE International Conference on Industrial Informatics (INDIN 2009), Cardiff, Wales, 2009.

- David A. McMeekin, Maja Hazdic, Elizabeth Chang, *Sequence Diagrams: An Aid for Digital Ecosystem Developers*, 3rd IEEE Digital Ecosystems and Technology (DEST'09), İstanbul, Turkey, 2009.

- David A. McMeekin, Brian R. von Konsky, Michael Robey, David J.A. Cooper, *The Significance of Participant Experience when Evaluating Software Inspection Techniques*, 20th Australian Software Engineering Conference (ASWEC '09), Gold Coast, Queensland, Australia, 2009.

- David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper, *Evaluating Software Inspection Cognition Levels Using Bloom's Taxonomy*, 22nd IEEE-CS Software Engineering Education and Training (CSEE&T 2009), Hyderabad, India, 2009.

- David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper, *Measuring Cognition Levels in Collaborative Processes for Software Engineering Code Inspections*, ICST I.T. Revolutions, Venice, Italy, 2008.

- David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper, *Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension*, 16th International Conference on Program Comprehension (ICPC 2008), Amsterdam, The Netherlands, 2008.

- David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, David J.A. Cooper, *Checklist Based Reading's (CBR) Influence on a Developer's Understanding*, 19th Australian Software Engineering Conference (ASWEC '08), Perth, Australia, 2008.

- David J. A. Cooper, Brian R. von Konsky, Michael C. Robey and David A. McMeekin, *Obstacles to Comprehension in Usage Based Reading*, 18th Australian Software Engineering Conference (ASWEC '07), Melbourne, Australia, 2007.

Table of Contents

Table of Figures

Table of Tables

List of Abbreviations used within this Thesis

CBR – Checklist Based Reading

UCR – Use Case Reading

UBR – Usage Based Reading

ADR – Abstraction Driven Reading

SBR – Scenario Based Reading

PBR – Perspective Based Reading

TBR – Traceability Based Reading

1.0 Chapter One

Introduction

One of the major motivations for the organizing of the conference was an awareness of the rapidly increasing importance of computer software systems in many activities of society (Software Engineering 1968).

1.1 Introduction

Approximately 70% of software budgets and resources are spent on maintaining existing software systems (Bennet 1990; Pfleeger Lawrence, & Atlee 2010; Swanson, & Beath 1989). Software developers attempting to understand the system they are working on consume a large portion of this time (De Lucia et al. 1996; Dunsmore et al. 2000; Hartzman, & Austin 1993). Therefore, both the ability to comprehend existing code and make effective changes is critical to the software industry. This thesis deals with software developer cognition levels. There are varying cognitive levels a developer experiences when executing different tasks within a software development project. In this thesis, the cognitive levels that developers demonstrate while performing software inspection related tasks are examined. These levels are examined in order to assist in increasing the ability of developers to understand and maintain a software system which they are commencing work on or returning to work on.

Sophisticated, complex software systems have become integral to most aspects of life in today's society. These systems are relied upon in order to deliver essential infrastructure and utilities, products and services. Manufactured products as well as infrastructure are often designed, produced and delivered through sophisticated and complex software systems.

The global financial system is largely computerised, relying solely on sophisticated software systems in order to function (Royal Academy of Engineering & British Computer Society 2004; Pressman 2005; Sommerville 2007; Kothari 2008). A problem in these computer systems can cause inconvenience to consumers as well as revenue loss to the supplier. The Commonwealth Bank of Australia has experienced major problems with their online banking system causing customers to lose access to their money via the Internet (Zappone 2009; 30 June 2009 news.com.au).

Software is considered a key factor and driving force into this century. Software now impacts upon almost every aspect of a person's life. They way we live, work, are educated and enjoy recreational activities has all been influenced by software (Royal Academy of Engineering & British Computer Society 2004). As a society, we have become dependent upon these software systems, and yet many of these systems contain interactions not well understood, that sometimes behave in unexpected ways (President's Information Technology Advisory Committee 1999). Two recent events have seen both Boeing and Airbus aircraft behave in unexpected ways due to software system malfunctions, causing injuries to passengers on board (Australian Transport Safety Bureau 2007; Australian Transport Safety Bureau 2009). Society is becoming more dependent upon systems that we understand less and less.

Software systems continue to increase in size and complexity as government, business, education and community organizations attempt to digitize their “essential services” in order to deliver goods and services in a more simplified and efficient manner. The IT infrastructure that delivers these services needs to be robust to support this. However, it is not only the hardware that must be robust. The software systems need to be reliable, robust and bug free.

Sommerville points out that “... as our ability to produce software has increased, so too has the complexity of the software systems that we need” (Sommerville 2007, p. 4). We now live what has been coined “The Digital Lifestyle” where so much of life is contained and retained in the digital zone: letters, photos, music, TV shows, movies, banking, communication, and more and more can be added to the list. On a daily basis people, create, share and interact through online communication media (Hofmann, & Thomas 2008).

Software developers are generally required to work with existing code bases (Sommerville 2007). For the developer to successfully maintain, correct and modify these systems, it is important that s/he understand the code base. These sophisticated and complex systems are often interconnected in such a way that they are tightly coupled. Changes to a single module may impact upon the remaining system in unpredictable ways.

Clients often have new requirements and request new features during the maintenance phase of software systems. This is generally due to the client’s

misconception about software flexibility and the ease with which new features can be incorporated and modification of existing features can be made (Royal Academy of Engineering & British Computer Society 2004).

Kothari says that the largest bottleneck preventing the continued advancement of these software systems is not in "...the time and space complexity, but the human complexity -- programmer's time and effort" (Kothari 2008, p. 3). Creating simpler systems is not the answer as the demand for even more complex systems continues to increase. Research into, and development of, ways to improve developers' abilities to understand and comprehend these systems is a possible solution to address Kothari's bottleneck.

This thesis examines the cognitive levels that developers express while performing various code inspection tasks. Understanding the expressed cognitive levels provides a way to map the different cognitive levels to the task the developer is performing. Through mapping and measuring these cognitive levels and observing the success of the task being performed, software inspection tasks can be created to focus specifically on increasing developers' comprehension and understanding of the system on which they are working. This enables the developers to write higher quality code and become more productive members within their software development team.

1.2 Software engineering and development

Software engineering is a term that was first used in 1968 at a conference sponsored by NATO Science Committee to address the crisis that loomed on the horizon with

the increased usage and reliance of computer systems in many aspects of society. With the changes occurring in society, computers were becoming more utilised and relied upon. It had been identified that the software development methodologies in use at that time were inadequate and there was a need to establish more rigorous software development processes. Taking engineering concepts from other engineering disciplines, the Software Engineering discipline was born and has continued to mature since that time.

A brief overview of different software engineering process models is given in the following sections.

1.2.1 Waterfall Model

An early software development model discussed in the literature is the Waterfall Model. The Waterfall Model is based on a traditional engineering process. Each stage within the cycle produces documentation that is signed off by the different stakeholders. The next stage of the process does not commence until the previous stage has been completed. Figure 1-1 shows the Waterfall model. When Royce (1970) first discussed the model, however, he did not refer to it as the Waterfall model. He pointed out that this model is one that should not be adopted in its current form, due to its limitations. He continued to describe the model using a feedback mechanism at each stage. Over time however, the model without the feedback mechanisms has become the canonical form with which other models are compared or from which they are derived (Pfleeger Lawrence, & Atlee 2010; Pressman 2005; Sommerville 2007).

Contained within the waterfall model are the four basic building blocks of every software development process. These building blocks are:

1. Software specification;
2. Software development;
3. Software Validation; and
4. Software evolution.

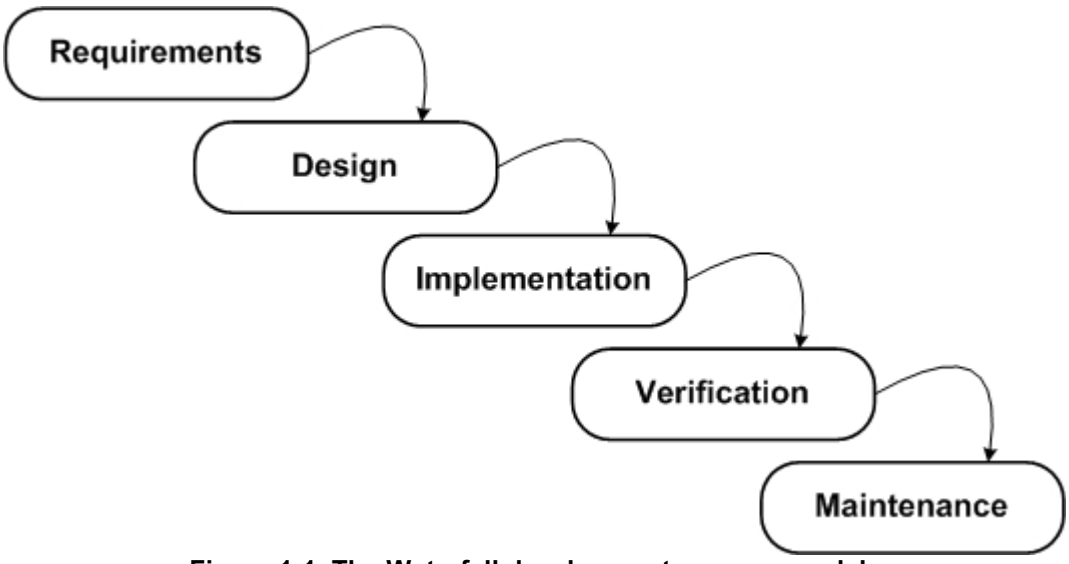


Figure 1-1. The Waterfall development process model.

The software specification building block is encapsulated within the Waterfall Model in the Requirements and Design bubble. The software development building block is encapsulated within the Implementation bubble. The software validation building block falls into the Verification bubble and the software evolution building block is encapsulated within the Maintenance bubble and can include software modification and correction.

Since the introduction of the Waterfall Model, several other models have been proposed. Waterfall with prototyping is such a model (Pfleeger Lawrence, & Atlee

2006). In this model, building a partial system allows stakeholders, the developer and customer, to examine the partially built system to ascertain whether the system will meet the customer's needs.

1.2.2 V-Model

Another adaptation of the Waterfall Model is the V-model depicted in Figure 1-2. This model was developed by the German Ministry of Defence. The model shows the direct links between the design and analysis of the system and the system testing and integration. The unit and integration testing are applied to verify that all stages of the program development have been correctly carried out. That is, the design and implementation are tested, not just the implementation (Pfleeger Lawrence, & Atlee 2010).

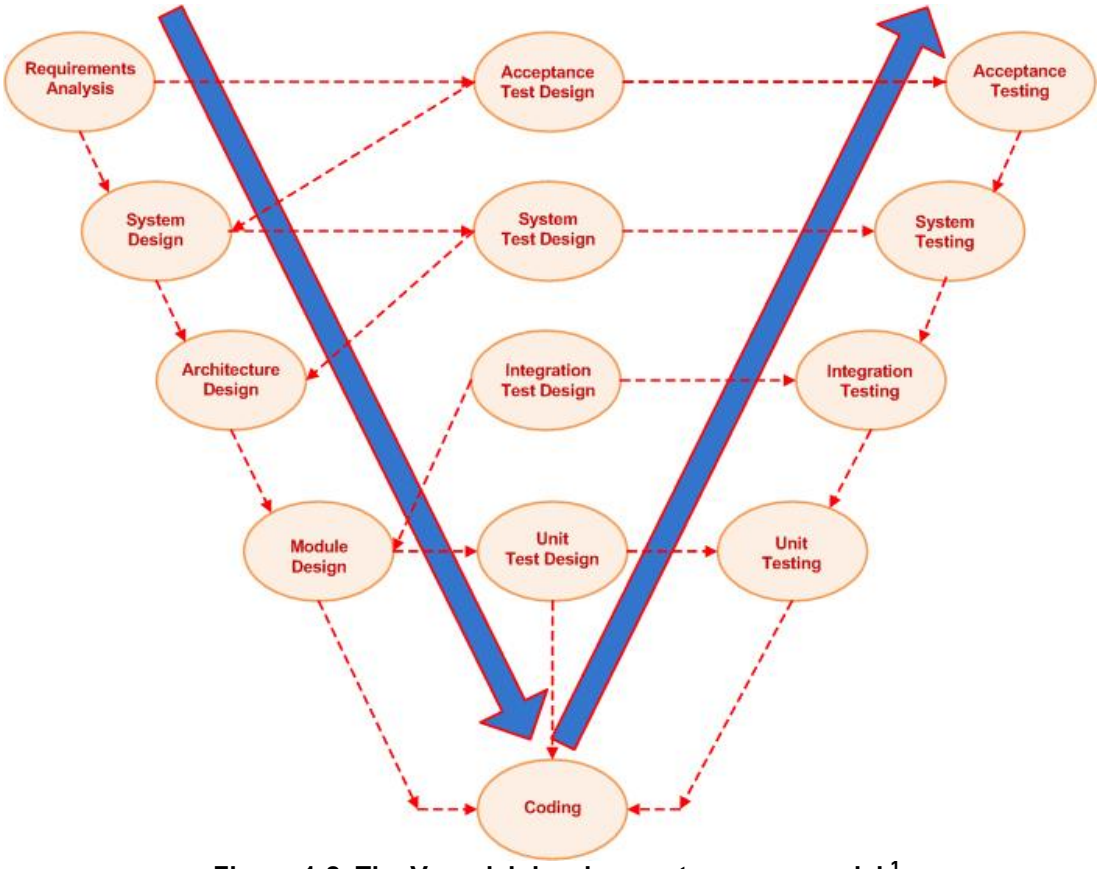


Figure 1-2. The V-model development process model.[1]

1.2.3 Prototyping Model

The Prototype Model is one that is often used within the context of other software development processes (Pressman 2005). However, it is also one that is used as stand-alone development process. This model is based on engineering prototyping modelling in which a system is quickly partially or fully built in order for stakeholders to gain a better understanding of the system (Pfleeger Lawrence, & Atlee 2010). From the prototype, the different stakeholders are able to identify whether or not the system is fulfilling their expectations and even if it is possible to create a system that fulfils expectations. Figure 1-3 depicts the cycle used within the software development process when the prototyping model is used.

[1] The image is in the public domain and downloaded available from Wikipedia.

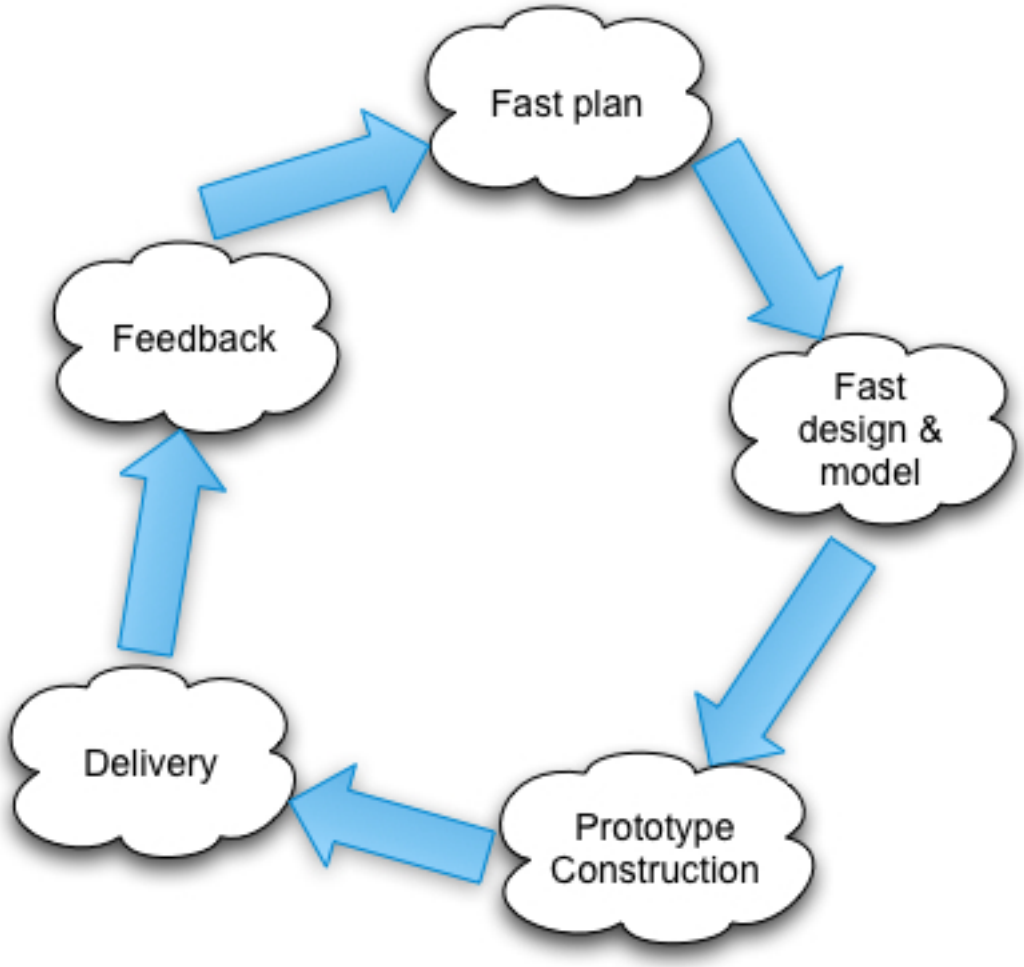


Figure 1-3. The prototyping development process model.

A caveat when applying the Prototyping Model is that once clients see the prototype, they often want to start using it immediately. However, it is important that developers generally not allow this as it is only a prototype. Allowing the client to use it can have negative consequences when the client encounters the prototype's shortcomings due to the rapid development cycle used to create the system.

1.2.4 The Spiral Model

Boehm (Boehm 1988) first proposed the Spiral Model of software development.

Boehm suggested that rather than representing the software development life cycle using a straight line, it should instead be seen as a spiral. In part, this is because in practical application some activities back track onto themselves. Each loop within the spiral is divided into four different sections:

1. Objective setting;
2. Risk assessment and reduction;
3. Development and validation; and
4. Planning.

One main difference between this development methodology and others was that it took into account the risk involved in each iteration within the spiral. That is, risk is explicitly identified with a detailed analysis conducted in order for this to be reduced within each iteration in the spiral.

Figure 1-4 depicts the spiral development process. Each quadrant within the spiral is focussed on certain tasks during the development process. Each iteration, labelled 1 – 4 in Figure 1-4 represents a new iteration through the development process. The innermost iteration focuses on gathering requirements from the client, keeping in mind the tasks of each quadrant. The second iteration may see the design being implemented. This continues and in each iteration more and more is added to the system.

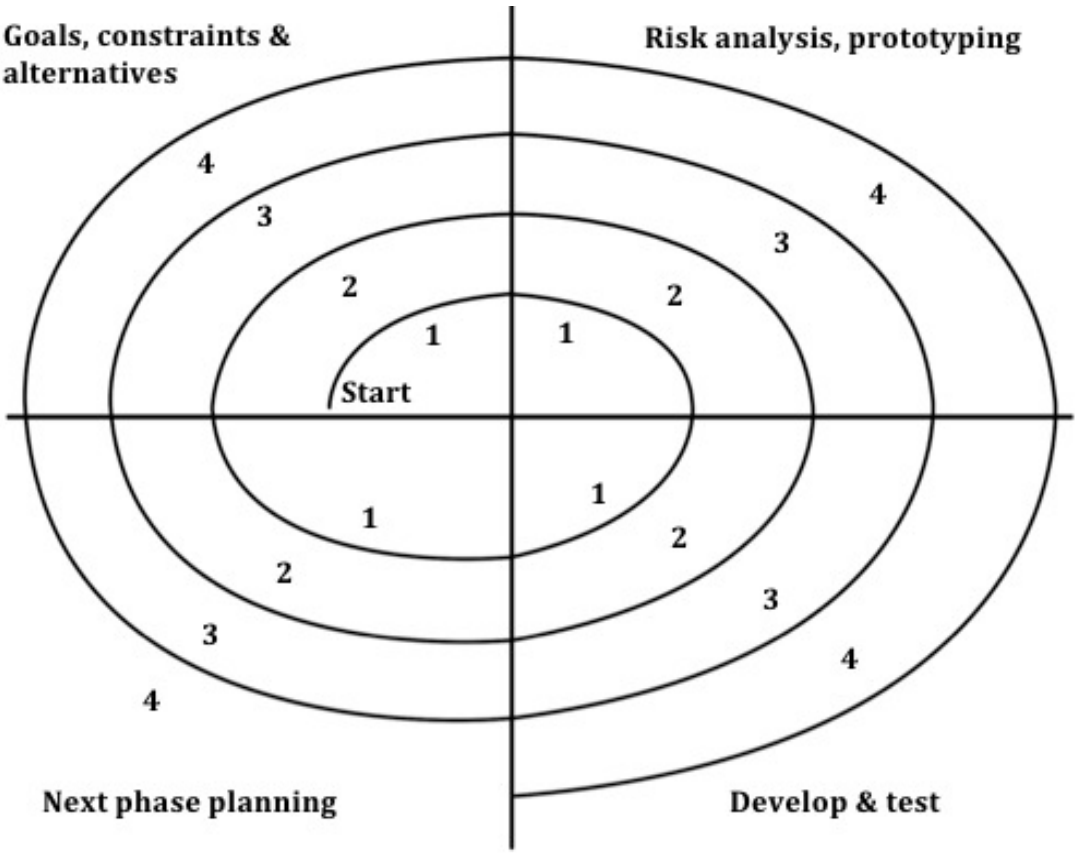


Figure 1-4.The Spiral development process model.

1.2.5 Agile Methods

Agile software development is not a single development process, but rather a name that groups together several development methodologies. Agile software development evolved from a group of 17 software engineering experts meeting to discuss "an alternative to documentation driven, heavyweight software development processes…" (Agile Manifesto 2001). There are many different methodologies grouped into this category. They include, but are not limited to: Extreme Programming (XP), Crystal, Feature Driven Development, Pragmatic Programming, Scrum and Adaptive Software Development (ASD).

1.2.6 Software development processes within this thesis

The different software development life cycle models explained previously highlight that the four basic building blocks of software development are implemented in each one.

This thesis relates to the fourth building block in the software development process: Software evolution/maintenance/modification/correction. Approximately 70% of a software product's life is spent undergoing maintenance (Bennet 1990; Pfleeger Lawrence, & Atlee 2010; Swanson, & Beath 1989) while between 50% and 90% of this time is consumed by a software developer attempting to understand the program in order to successfully make the necessary changes to the system (De Lucia et al. 1996).

Sommerville (Sommerville 2007) pointed out that very rarely do developers start building software from scratch when they are new to a team. Instead, they start with an already existing code base and develop from that. Therefore, it is within the fourth building block that a vast majority of software developers must learn about the system on which they will be working. Even though this seems obvious a certain level of learning and understanding is necessary, before they commence. The software development life cycle models do not explicitly cater for this as a preliminary stage through which most new developers must progress.

This thesis identifies methodologies that software developers can apply to aid them to more effectively understand the software system they are working on, within the bounds of the fourth building block. The goal is to increase the efficiency of the time

invested to understand the system so that they can successfully maintain the system.

1.3 Software inspections

Software inspections are review processes implemented to detect defects in the software artefacts under inspection (Fagan 1976; Shull et al. 2001). Software inspections are implemented early in the development process and offer developers a structured method to examine software artefacts for defects (Fagan 1976; Dunsmore et al. 2003). Software inspections and their success in detecting defects is a well-researched area in software engineering (Sjøberg et al. 2005).

A defect is defined as "a deviation from the specification that would go on to cause failure (some undesired behaviour of either a trivial or catastrophic nature) if left uncorrected" (Dunsmore, 2002). Inspections are considered as one of the most effective software engineering practices to ensure quality (Laitenberger, & DeBaud 2000). Since their formalisation, inspections have been shown to reduce defect numbers in both design and code, reduce testing time, and improve production cost effectiveness (Fagan 1976; Fagan 1986; Travassos et al. 1999; Laitenberger, & DeBaud 2000; Humphrey 2000). Nevertheless, misunderstandings often occur when using the term 'software inspection'.

Software inspections are a practical methodology widely used in the ICT industry to identify defects early in the development life cycle (Fagan 1976). Interactive Development Environments (IDEs) such as Netbeans, Eclipse, Xcode and Visual Studio, with their auto completion functions, warn developers of many potential errors before they finish 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.

Traditionally, software inspections have been applied specifically to detect defects within software artefacts. Figure 1-5 illustrates inputs and outputs to the software inspection process. There are three different inputs into the software inspection process: the code and/or artefact being inspected, the inspection technique that will be applied during the inspection and the inspector(s) experience level. The experience level includes experience as a software developer, as a software inspector and with the code and/or artefact being inspected. The inspection results in a defect list being created, which also may contain false positives. The code is then reworked and inspected again. In an ideal world, this cycle would continue until all defects have been removed. However, depending upon the company, this cycle usually has a set number of iterations.

Reading software artefacts is an essential practice for producing high-quality software during a product's development and maintenance life cycle (Basili 1997). Inspection techniques/reading strategies are usually linked with verification and validation of software artefacts. Applying software inspections to improve a developer's ability to understand and maintain a code base is an area that has not been well researched as an additional and possibly significant benefit to the software inspection process.

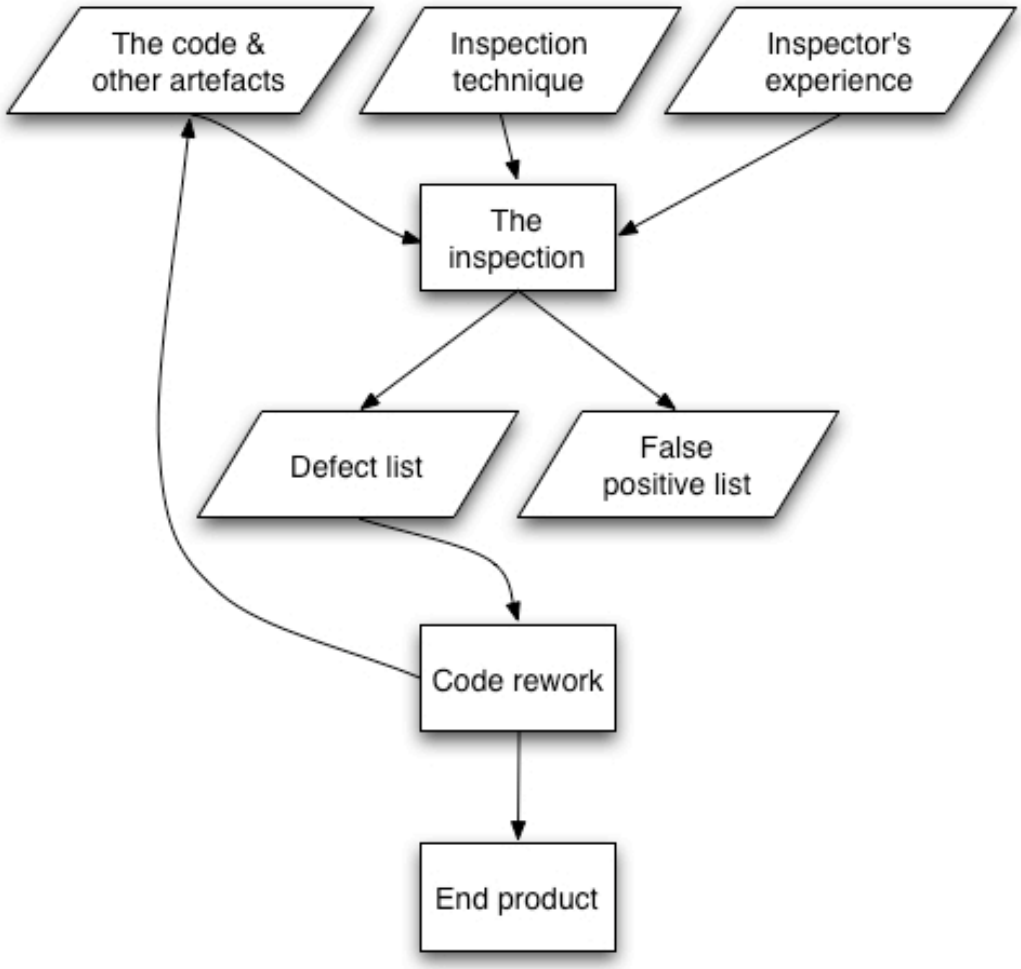


Figure 1-5 Traditional inspection process.

1.4 Cognitive models of developer understanding

"Computer programming is a human activity" (Weinberg 1971). The sophisticated complex programs mentioned early in this chapter are still designed and created by humans. Computer programming has become extremely important within today's world as so much now depends upon software to function.

How humans go about programming has been researched from early in the history of the computing field. Weinberg's *The Psychology of Computer Programming* (Weinberg 1971), Shneiderman's *Software Psychology: Human Factors in*

Computer and Information Systems (Shneiderman 1980) are two classic examples demonstrating that computer programming is a human activity.

A vast amount of research has been conducted to understand the cognitive processes that developers use as they attempt to understand the software system that they are working on. Much of the research focus has been on how programmers develop their program understanding, their mental processes, and the cognitive models they use to acquire the necessary understanding in order to effectively carry out the task required of them.

Five cognitive processes have been identified that characterise how developers go about understanding the program they are working on. These are:

1. Bottom-up;
2. Top-down;
3. Systematic;
4. As needed; and
5. Integrated.

However, very little has been written about methodologies and techniques that can help developers to understand the techniques they are using and even increase their efficacy. This thesis examines software inspection techniques and the cognitive processes underlying these techniques. The thesis will then look at specific software inspection techniques and the cognitive levels that developers express while applying those techniques.

Based on this new knowledge, a reading methodology will be proposed. The reading methodology will: 1) enable team members to be more effective code maintainers, 2) take into account developer experience, and 3) continue to enable defects to be detected.

A reading methodology will also be proposed for use within Software Engineering, Computer Science, Information Technology and Information Systems degree courses. The methodology will recommend the software inspection reading techniques to be taught at specific parts of the courses with a justification for each technique.

1.5 Bloom's Taxonomy - Cognitive Domain

In 1956, a group of educators in the United States of America, led by Benjamin Bloom, built a taxonomy of educational objectives. The goal was to assist educators as they built curriculums. This taxonomy would aid educators to specify learning objectives so the learning experience could be created and then evaluated (1956).

The taxonomy's aim was to provide a classification for the educational objectives of tasks set out within a learning curriculum. The taxonomy starts with simple knowledge recall and moves through to the higher cognitive levels of evaluation and synthesis. Bloom's Taxonomy, as it is now known, has been and continues to be used throughout the world within the education field. The six categories cited from Bloom (1956) are: Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation.

Within the context of this thesis, Bloom's Taxonomy is used as a way to categorise the cognitive levels that software inspectors express as they perform the tasks required of them. Effectively, the goal is to develop inspection strategies that will enable developers to maintain code, which requires Evaluation and Synthesis skills.

1.6 Scope of this thesis

This thesis presents a methodology for conducting software inspection activities in order to assist developers to operate at higher cognitive levels as described within Bloom's Taxonomy. This is achieved through two different avenues: the examination and categorisation of the cognitive levels developers expressed as they performed different software engineering inspection related tasks, and mapping different software inspection techniques into the various cognitive process models described within the literature. This mapping is then followed by a mapping of these inspection techniques into the Bloom's Taxonomy levels.

This thesis does not examine the ways in which developers go about understanding the code. Rather, it analyses the developers' verbalised expressions and categorises them into appropriate cognition levels, as described by Bloom's Taxonomy.

70% of the development budget for a software system is consumed by the maintenance phase and between 50% and 90% of this time is used by developers attempting to understand the program they are maintaining. The creation of a methodology to improve developers' efficacy in understanding a system would have a significant impact on the product development's budget of time and resources. Due

to time and budget resource limitations for this PhD research, this thesis does not examine, nor attempt to examine, the financial or time savings that would be gained through the application of this methodology. This is reserved for future work.

The thesis presents a proposed methodology that can be applied to assist developers to operate at the higher cognitive levels as set out in Bloom's Taxonomy. The reading methodologies can be implemented within Software Engineering, Computer Science, Information Technology and Information Systems degree programs as well as in industry-based training courses. Facilitating developers to operate at higher levels of thinking enables the developer to grasp and understand the software system on which they are working, knowing the implications of the changes they make to the system. A greater understanding of the system, what it does and how it goes about doing it, provides the developer with the ability to carry out the required maintenance and evolutionary tasks with more accuracy and reduce the possibility of introducing new defects.

1.7 Motivation/Objectives of the research

In the previous sections, the important role that software plays in today's society has been identified. Examples were given of the software development processes used to create this software and the challenge developers face in understanding the software system on which they are working. Within each software development process model, there is no explicit task or time set aside for developers to learn the system. Learning and understanding the system is an assumed process and yet this has been identified as one of the major bottlenecks in the advancement of software development. This thesis proposes ways that will assist developers to more

effectively learn and understand the existing system on which they are working in order to maintain code, resulting in shorter maintenance cycles with fewer defects. This thesis examines a possible significant secondary benefit of software inspections. Specifically, to what extent do software inspections improve an inspector's knowledge of the system and overall understanding of the code under inspection through facilitating higher order thinking? Does participation in an inspection improve the effectiveness of subsequent synthesis and evaluation tasks, such as adding new functionality during software maintenance? To what extent is this influenced by the inspection technique utilized and the level of experience of the inspector?

In order to achieve this, the research has been broken down into several objectives:

Objective One: understand the significance and impact of experience when a software developer carries out software inspections. This understanding will enable the construction of empirical research based on both task and participant experience level.

Objective Two: identify if there is correlation between the number of defects detected by an inspector and the number of successful modifications s/he makes to the inspected code, when adding new functionality to the code base. Identifying this will form a basis for understanding whether software inspections can be used to facilitate increased understanding of the software system on which they are working.

Objective Three: identify the cognitive levels at which inspectors operate while

performing a software inspection using different software inspection techniques. An understanding of this will facilitate an understanding of the differences between software inspection techniques and the cognitive levels they assist in facilitating. From this understanding, the adaption and application of different software inspection techniques for developers with different levels of experience will be possible.

Objective Four: map the different software inspection techniques to the different cognitive models described within the literature and map the cognitive models and inspection techniques into Bloom's Taxonomy of Learning - Cognitive Domain. This will facilitate the classification of software inspection tasks in teaching and training programs in order to aid developers to function at the higher cognitive levels.

Objective Five: propose a code reading methodology that allows developers, according to their experience levels, to operate at the higher cognitive levels when they are carrying out software engineering tasks upon a system's code base.

1.8 Significance of this research

The contribution of this thesis is in several areas: first, the identification of the differences between novice developers in comparison to professional developers as participants in software inspections, and how experience levels impact upon the performance and perception of software inspection techniques. Second, the mapping of different inspection techniques to cognitive models will provide a way for users of these approaches to better understand what they are doing and how this may affect

their understanding of the system. Third, identifying the inspection techniques with their respective cognitive models and showing which level of Bloom's Taxonomy it correlates with, will aid software project managers to ascertain the cognition level at which developers are likely to be operating when they are performing a task. It will provide software development team managers with a reading strategy they can deploy to increase the learning efficacy of new team members. It also provides a way for Software Engineering, Computer Science, Information Technology and Information Systems course curriculums to be designed with different tasks built into them knowing the level of Bloom's Taxonomy that is being taught and examined. This is particularly important as, at university level, the teaching methodology and the vast majority of the course content should require students to function at the higher cognitive levels (2009).

At the early stages of degree programs, there is a much stronger emphasis on lower levels of thinking, giving students the foundation of knowledge needed to become professionals in their areas. However, if while this lower level knowledge is being delivered, students are also being provided with ways to facilitate higher thinking levels, then their learning experience can more readily take them into the higher cognitive levels. This in turn is building the skills that will be required of them when they are working as industry-based professionals, when called upon to evaluate existing systems as well as design and create extensions to existing systems or completely new systems.

1.9 Thesis structure

This thesis is structured as follows:

Chapter 1: provides an overall introduction to the areas that will be addressed within this thesis.

Chapter 2: provides a literature review in several areas of software engineering that are related to the work to be conducted within this thesis. These areas include software inspections and software inspection techniques, program comprehension, reverse engineering and visualisation, delocalisation, Bloom's Taxonomy and the Think-aloud protocol.

Chapter 3: defines the problems that this thesis will address. The problem definition will be broken down into the different research areas to be addressed by this thesis. The chapter will also provide definitions of key terminology that will be used throughout this thesis as well as discuss the different research methodologies available for use in a thesis such as this. The choices made regarding the pertinent research methodology that has been applied within this thesis will also be identified.

Chapter 4: will provide an overview highlighting and outlining the solutions that will be proposed to the problems identified in Chapter Three. A map for the remaining chapters of the thesis describing how each contributes to the solution is also provided and described within this chapter.

Chapter 5: examines the impact that a software developer's experience has upon the performance and perception of a software inspection and the technique implemented within the inspection. The Checklist-Based Reading, Use-Case Reading and Usage-Based Reading techniques will be compared with both student and industry

professional developers.

Chapter 6: looks at the cognitive levels that developers express as they carry out a Usage-Based Reading task to map a use case scenario, shown in a sequence diagram, to the underlying executing code. *Bloom's Taxonomy of Educational Objectives, Cognitive Domain* (1956), in conjunction with the Context-Aware Analysis Schema (Kelly, & Buckley 2006) is introduced in order to measure developers' expressed cognitive levels as they carry out the UBR task.

Chapter 7: investigates the impact of software inspection for non-traditional purposes. Based on the findings from the previous chapter, this chapter examines whether a correlation exists between the number of defects detected during an inspection and a developer's ability to maintain code. This is measured in an experiment in which inspectors are asked to add new functionality to the code they have inspected.

Chapter 8: this chapter examines the cognitive levels that developers express while they carry out a code inspection implementing different techniques. The expressed levels are to be examined in order to understand the cognitive levels that are facilitated by the different software inspection techniques tested,

Chapter 9: this chapter further examines the expressed cognitive levels of developers as they carry out a task to add new functionality to the previously inspected code base. It examines the different cognitive levels in relation to the inspection technique the inspector implemented and the cognitive levels that were

facilitated during the inspection.

Chapter 10: this chapter examines the software inspection techniques used within this thesis. It presents a mapping of the inspection techniques to the cognitive process models the inspection technique facilitates the inspector to implement. It discusses the Bloom's Taxonomy cognitive level that the inspection technique maps to, as derived from the way in which it has inspectors carry out the inspection task and also from the results reported within the experiments.

Chapter 11: provides a set of guidelines for the implementation of software inspection techniques as a technology to assist developers to operate at higher cognitive levels. Recommendations are given for the introduction and implementation of software inspections as a reading technology within both degree programs and training programs for software developers.

Chapter 12: concludes this thesis with a summary of the work that has been done and the results that have arisen from it. It then identifies remaining and future work within this field as seen by the author.

1.10 Conclusion

This chapter has provided an introduction to the complexity of software and the sophisticated systems used throughout the world today, and the impact this can have when those systems fail. Software development methods and processes have been introduced along with the cognitive process models that developers use when attempting to understand a software system. Software inspections and the role they

have played in detecting defects was touched upon, recognising that the application of software inspections as a reading technique for improving developer program comprehension and understanding is an area that has not yet been researched.

The next chapter presents an overview of the existing literature pertinent to this research: software inspections, program comprehension, reverse engineering, visualisation, Bloom's Taxonomy, think-aloud protocol and students in software engineering empirical research. This review is conducted firstly, to ensure that this work has not been carried out previously by another researcher, and secondly to establish the place and relevance of this research within the wider body of knowledge.

Bibliography

Agile Manifesto, 2001, Manifesto for agile software development, viewed 8 February 2010, http://www.agilemanifesto.org

Australian Transport Safety Bureau (ATSB), 2007, *In-flight upset event 240km north-west of Perth, WA Boeing Company 777-200, 9M-MRG 1 August 2005*, Aviation Occurrence Report - 200503722.

Australian Transport Safety Bureau (ATSB), 2009, *In-flight upset 154km west of Learmonth, WA 7 October 2008 VH-QPA Airbus A330-303*, Aviation Occurrence Report – AO-2008-070.

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

Bennet, K.H., 1990, An introduction to software maintenance, *Information and Software Technology*, 12(4), pp. 257-64.

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

Boehm, B.W., 1988, A spiral model of software development and enhancement, *Computer*, 21(5), pp. 61-72.

Dunsmore A. *Investigating effective inspection of object-oriented code* (2002).

Dunsmore, A., Roper, M. & Wood, M., 2000, The Role of Comprehension in Software Inspection, *The Journal of Systems and Software*, 52(2--3), pp. 121-9.

Dunsmore, A., Roper, M. & Wood, M., 2003, The development and evaluation of three diverse techniques for object-orientated code inspection, *IEEE Transactions on Software Engineering*, 29(8), pp. 677-86.

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

Fagan, M.E., 1986, Advances in Software inspections, *IEEE Transactions on Software Engineering*, 12(7), pp. 744-51.

Hartzman, C.S. & Austin, C.F., 1993, Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative researchCASCON '93, *Maintenance productivity: observations based on an experience in a large system environment.* IBM Press, pp. 138-70.

Hofmann, G.A. & Thomas, G., 2008, Digital Lifestyle 2020, *Multimedia, IEEE*, 15(2), pp. 4-7.

Humphrey, W.S., 2000, *Introduction to the team software process,* Addison--Wesley, Massachusetts.

Kelly, T. & Buckley, J., 2006, 14th IEEE International Conference on Program Comprehension (ICPC2006), *A Context-Aware Analysis Scheme for Bloom's Taxonomy.* pp. 275-84.

Kothari, S.C., 2008, Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, *Scalable Program Comprehension for Analyzing Complex Defects.* pp. 3-4.

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

De Lucia, A., Fasolino, A.R. & Munro, M., 1996, Proceedings of Fourth Workshop on Program Comprehension, 1996, *Understanding function behaviors through program slicing.* pp. 9-18.

NetBank fails on last days of financial year, 2009, *news.com.au.* Retrieved July 5, 2009, from http://www.news.com.au/technology/story/0,28348,25712292-5014239,00.html

Pfleeger Lawrence, S. & Atlee, J.M., 2006, *Software Engineering: Theory and*

Practice , Pearson/Prentice Hall, Upper Saddle River, N.J..

Pfleeger Lawrence, S. & Atlee, J.M., 2010, *Software Engineering Theory and Practice,* Fourth (International) ed. Pearson, Upper Saddle River, N.J..

President's Information Technology Advisory Committee (PITAC), 1999, *Information Technology Research: Investing in our future*, Arlington VA, U.S.A.

Pressman, R.S., 2005, *Software Engineering A practitioner's approach,* Sixth ed. McGraw--Hill, Boston,USA.

Royal Academy of Engineering & British Computer Society, 2004, *The Challenges of Complex IT Projects*, The Royal Academy of Engineering, London, U.K.

Royce, W.W., 1970, IEEE WESCON, *Managing the Development of Large Software Systems.* IEEE, pp. 328-38.

Shneiderman, B., 1980, *Software psychology: Human factors in computer and information systems,* Winthrop Publishers.

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

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

Software Engineering, 1968, Naur, P., & Randell, B. eds. NATO Scientific Affairs Division, Garmisch, Germany.

Sommerville, I., 2007, Inaugural IEEE Conference on Digital EcoSystems and Technologies, DEST '07, 2007, *Design for failure: Software Challenges of Digital Ecosystems.* IEEE Computer Society.

Swanson, E.B. & Beath, C.M., 1989, *Maintaining information systems in organizations*.

Teaching and Learning at Curtin, 2009, Office of Teaching and Learning, Curtin University of Technology.

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

Weinberg, G.M., 1971, *The psychology of computer programming,* Dorset House Publishing.

Zappone, C., 2009, CBA struggles to fix NetBank, *WAtoday.com.au.* Retrieved July 5, 2009, from http://business.watoday.com.au/business/cba-struggles-to-fix-netbank-20090701-d4gk.html

2.0 Chapter Two

Literature Review

A discipline comes of age when it seriously contemplates its own past.

E.F.K. Koerner (Koerner 1999).

2.1 Introduction

In this chapter, an overview of the existing literature and work related to this thesis is presented. The different literature fields surveyed include software inspection, program comprehension, reverse engineering and visualisation, delocalisation, protocol analysis, learning classification taxonomies and the different participant experience levels employed within software engineering empirical research. The existing literature is the foundation upon which this research is based. The analysis highlights the need for the research contribution subsequently described within this thesis.

The first two sections examine software inspections and the different techniques: their goals, methodologies and applications. The third section examines program comprehension. This includes a description of program comprehension and the different comprehension strategies and methodologies identified within the literature. Reverse engineering and software visualisation is discussed followed by an examination of delocalisation, a phenomenon that Object Oriented programming

has exacerbated. Learning classifications are expanded together with the way they have been previously used within Software Engineering, Computer Science, Information Technology and Information Systems. Protocol Analysis or think-aloud data collection and the way it has been used within software engineering and software engineering education will then be examined. The final section of the literature review examines the impact of using both student and professional participants in software engineering empirical research.

The literature reviewed will then be summarised, establishing the foundation of previous work while identifying areas that require further research.

2.2 Software Inspections

Software inspections are a review process implemented to detect defects in the software artefacts under inspection (Fagan 1976; Travassos et al. 1999; Shull et al. 2001; Dunsmore et al. 2003). "Debugging is a cognitively demanding problem-solving process, which requires both appropriate knowledge and reasoning skills" (Yoon, & Garcia 1998, p. 160). Software inspections are one approach that can be applied to assist in the problem-solving process.

McConnell (McConnell 2004) states that up to 60% of defects are detected and then removed through individual inspections, and claims inspection to be the most effective defect removal tool, bar perhaps prototyping and beta testing in high volume.

Software inspections started as a group activity (Fagan 1976) and have evolved over

time to be a much more individual activity. This is due largely to the empirical evidence indicating that group meetings did not discover significantly greater numbers of new defects than those already discovered individually (Porter et al. 1995; Votta Jr 1993; McCarthy et al. 1995). Eick et al. (Eick et al. 1992) highlight that without a meeting, inspectors may not review the artefact as thoroughly because there is no peer group as there would be if they were required to present defects they had identified to a group of inspectors.

The term 'software inspection' often means different things to different people. There are usually four terms used interchangeably when people consider software inspections. They are:

1. Inspection
2. Review
3. Technical review
4. Walk-through

However, according to IEEE Standard 1028-2008 (2008) these four terms have distinct meanings:

- **Inspection**: is "a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications."
- **Review**: is "a process or meeting during which a software product, set of software products, or a software process is presented to project personnel, managers, users, customers, user representatives, auditors or other interested parties for examination, comment or approval."

- **Technical review**: is “a systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards.”
- **Walk-through**: is “a static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible anomalies, violation of development standards, and other problems.”

Software inspection processes are specifically designed to detect defects within the system according to the IEEE Standard 1028-2008 definitions. The remaining three processes are designed for evaluation purposes (Aurum et al. 2002).

Software inspections have existed since the early days of software development. Babbage, considered by many to be the first software developer, had colleagues including Ada Lovelace inspect the programs he wrote. Von Neuman, the father of modern computer architecture, sought out his colleagues to inspect the programs he Awrote (Knight, & Myers 1993). Using modern nomenclature, these would be classified as informal inspections. However, they were inspections nonetheless.

Historically, the anecdotal evidence suggests that the forefathers of software development simply assumed the need for colleagues to inspect the software they developed, and hence made very little note of it in their writings (Weinberg, & Freedman 1984).

As the 1950s were coming to an end, software development managers were seeing the need for formal inspections. During the 1960s, this process began to be formalised (Weinberg, & Freedman 1984).

Fagan's article *Design and code inspections to reduce errors in program development* was one of the seminal papers in the formalisation of inspections as a software engineering process (Fagan 1976). Inspections were defined as "a formal, efficient, and economical method of finding errors in design and code" (Fagan 1976, p. 189). Fagan reported that in some cases, up to 85% of errors were detected through inspections.

Inspections are considered as one of the most effective software engineering practices to ensure quality (Laitenberger, & DeBaud 2000). Since their formalisation, inspections have been shown to reduce defect numbers in both design and code, reduce testing time, and improve production cost effectiveness (Fagan 1976; Fagan 1986; Travassos et al. 1999; Laitenberger, & DeBaud 2000; Humphrey 2000).

Fagan (1976) reports that detecting defects in the first half of a project is between 10 and 100 times cheaper to fix than if the defects were detected in the second half of the project. This reduction in re-work flows on, leading to increased productivity by the developers. The time-saving was calculated from early detection that led to a 23% productivity increase by the coders.

IBM formally introduced software inspections into their software development processes in 1974. In the following ten years they reported doubling the number of lines of code delivered, while reducing defects by two-thirds (Fagan 1986).

Jones (Jones 1996) reports that the combination of design inspections with code inspections removes up to 85% of defects within a software project prior to release.

In as much as inspections have been reported as successful, Laitenberger and DeBaud (2000) highlight that inspections have not fully impacted upon and permeated the software development industry. They highlight that much of the work published has not been made into a "coherent body of knowledge taken from a life-cycle point of view" (2000, p. 6). This may be why inspections do not occur within all software development projects. Yet Sjøberg et al. (2005) found that 33% of studies within the software engineering literature reported within the ten years from 1993 to 2002 contained software inspections.

Even though software inspections have not been applied within all software development projects, they are still considered a standard process in the software development life cycle. The "Fagan Inspection" has become the de facto inspection technique (Laitenberger, & DeBaud 2000; Tyran, & George 2002).

2.2.1 Fagan Inspection

The "Fagan Inspection" team was made up of four persons, each with a specific role, Table 2-1 shows each role accompanied by a brief description (Fagan 1976, p. 190).

Table 2-1 Four roles in the Fagan Inspection.

Role	Description
Moderator	Moderates and manages the inspection team offering it leadership.
Designer	The system designer.
Code Implementer	The programmer who wrote the code.
Tester	The individual responsible for designing and or implementing the test cases.

The five steps involved in the process were (Fagan 1976, pp. 192-194):

1. **Overview**: the designer gives to the inspection team a general overview of the area being inspected.
2. **Preparation**: the inspection team does their homework, attempting to understand the design, with some errors being found during this time.
3. **Inspection**: the team meets as a group and a reader, selected by the moderator but usually the designer, describes how he will implement the design.
4. **Rework**: the designer addresses the errors and problems identified in the previous step.
5. **Follow-up**: the moderator follows up the re-work here to ensure that all issues raised in step three have been addressed.

2.3 Reading software code

Software inspections are code reading activities. Inspection or reading techniques are a tool provided for inspectors in order to increase their effectiveness when reading software artefacts to detect more defects (Travassos et al. 1999). Reading software artefacts is an essential and indispensable technology used to produce high-quality software and is the only technology available throughout the entire development and

maintenance process (Basili 1997). Basili also highlights that reading software as a skill, and the technologies to facilitate this, have been vastly ignored even though reading is a central activity/technology in verifying and validating whether a software product performs correctly (Basili et al. 1996; Basili 1997). Additionally, software inspections assist inspectors to develop greater insight into, and understanding of, the artefact being inspected (Laitenberger, & DeBaud 2000; Kelly, & Buckley 2006).

Systematic, accurate reading of software is a vital skill for the software developer and is the cornerstone to validating and modifying systems written by others (Linger et al. 1979). In most other areas of life, we first learn to read prior to learning to write (Basili 1997; Gabriel, & Goldman 2000). Other disciplines learn by reading the works of others. Writers read other writers' great works of literature; engineers learn from the critical analysis of others' works, academic supervisors advise their students to read others' theses.

In software development, we learn to write programs but generally not how to read them, and developers are not specifically encouraged to read the code of others as a learning experience. It must be noted that reading the code of some programs would actually require one to break the law of several countries in the reverse engineering of these products. There has been some work on code reading as a skill carried out by Spinellis (Spinellis 2003) as well as by the Carnegie Mellon Institute (Deimel & Naveda 1990).

Research that compared software inspectors with software testers showed that upon

completion of their tasks, the inspectors had a better understanding of the software, what and how it performed, than did the testers. The software inspectors better estimated the number of defects that remained within the system than did the testers (Basili 1997).

2.4 Inspection Techniques

A software inspection technique, often referred to as a reading technique, is a step-by-step procedure that guides a software inspector through a software artefact(s) as s/he searches for defects. A reading technique is a tool that inspectors use to detect defects (Laitenberger, & DeBaud 2000). These techniques enable well-defined, systematic artefact inspection strategies that provide both feedback and correction, thereby improving final product quality (Shull et al. 2000).

Within the literature many different inspection techniques have been described. Each inspection technique differs in some way from the others. The following sections will outline several of the inspection techniques contained in the literature, describing their application, strengths and weaknesses.

2.4.1 Ad hoc

The Ad-hoc technique is an informal but common inspection technique (Laitenberger, & DeBaud 2000). It is considered one of the simplest inspection techniques to implement, as the technique has no formal methodology, 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,

systematic inspection of the artefact using personal experience and understanding as the guide (Laitenberger, & DeBaud 2000). Even though there are no guidelines for this method, it is still regarded as a reading technique (Porter et al. 1996; Laitenberger, & DeBaud 2000) and is considered effective in detecting defects.

A strength of the Ad-hoc technique is that it gives the experienced developer freedom to read the code and execute the inspection according to his/her own experience.

A weakness of the Ad-hoc technique is its lack of formal process: the novice developer may lack the necessary experience to effectively carry out an Ad-hoc software inspection (Dunsmore et al. 2002; Laitenberger, & DeBaud 2000). Hence, upon completing the inspection, the novice developer's results may not accurately reflect the current state of the software. The novice developer may not have the experience necessary to successfully apply this inspection technique.

2.4.2 Checklist-Based Reading (CBR)

The Checklist-Based Reading (CBR) technique is a structured systematic methodology for inspecting software artefacts. The CBR technique is considered the seminal software inspection technique (Aurum et al. 2002). CBR is the most commonly used inspection methodology (Tyran, & George 2002), is regarded as the standard inspection technique in many software development organisations throughout the world (Laitenberger, & DeBaud 2000), and is considered the benchmark by which other inspection techniques should be measured (Thelin et al. 2003).

Each software artefact has its own unique checklist. The questions that make up each checklist are created based on the source of defects identified in an organisation's historical data. In this way, the questions reflect problematic issues that the organisation has previously encountered in earlier systems (Gilb, & Graham 1993; Humphrey 1995).

According to Brykczynski (1999), the checklist should fit on one side of a single sheet of paper. This prevents an inspector from having to continuously turn pages back and forth, thus interrupting their concentration on the task at hand.

Feature	No.	Question	Y	N	Defect Ref.
For each class:					
Class Fields	1.	Is each class field declared private, protected, or public as appropriate?			
For each method:					
Method	10.	Are the numbers of parameters correct for each method?			

Figure 2-1. An extract from a checklist.

In CBR, the inspector answers a series of questions regarding the artefact under inspection. Each question requires either a "yes" or "no" answer. A yes answer indicates the artefact, in relation to that question, is defect free. A no answer indicates the possibility of a defect in the artefact, requiring the inspector to carry out further investigation. Figure 2-1 is an extract from a typical checklist.

The strengths of the CBR technique are that it explicitly directs the inspector as they

search for defects within the artefact. The checklist is a product of prior inspections, therefore capturing organisational history with respect to prior defects.

The limitations of the CBR technique are due to the highly structured process that can impede inspectors, especially experienced ones, from reading the code in a more natural manner. Additionally, inexperienced inspectors may be directed away from defects not directly addressed by checklist questions and hence these defects will not be detected.

2.4.3 Abstraction-Driven Reading (ADR)

The Abstraction-Driven Reading (ADR) technique (Dunsmore et al. 2003), originally called Systematic Inspection (Dunsmore et al. 2001), was first described as a "systematic, abstraction-based reading strategy for object oriented code inspection" (Dunsmore et al. 2002, p. 57). It was created because object-oriented languages can often create a delocalisation effect where related code is spread throughout the system. Small methods, polymorphism, inheritance and dynamic binding often cause this delocalisation effect. ADR attempts to "localise the de-localisation."

At the completion of the ADR inspection, a natural language abstraction is written by the inspector for each inspected method, highlighting changes in an object's state by identifying changes to the attributes, class fields, and its outputs or return values. The abstracts should be compendious, declarative and complete.

In order to do this, the inspector reads the code in a systematic manner. Ideally,

classes are ordered so that the class with the least number of dependencies is inspected first and that with the most dependencies is inspected last. Inside each class, methods are inspected in a similar manner. The method with the least number of dependencies is inspected first, continuing on to the one with the highest number of dependencies.

Where necessary, the inspector follows calls to external sources, also inspecting the invoked code for understanding. As the inspector reads the code, a list of identified defects is generated. The process continues in this manner until the specified code is completely inspected.

The ADR technique is similar to the Step-Wise Abstraction technique described by Basili (Basili 1997) but originally from Linger et al. (Linger et al. 1979).

By creating the natural language abstracts, it is thought that this will force a deeper, more thorough understanding of the code and system (Dunsmore et al. 2001; Dunsmore et al. 2003). The abstracts created can also be reused later, providing those uses with accurate descriptions of the classes and their methods. However, this technique comes with concomitant costs in time, and can be cognitively overwhelming as the inspector follows calls to many and various areas of the code and then must attempt to write the descriptive abstracts. Also, the inspector may end up spending large amounts of time examining code outside the scope of the inspection.

2.4.4 Use Case Reading (UCR)

The Use Case Reading (UCR) technique evolved to aid inspectors of Object Oriented systems as they encountered the dynamic interaction of collaborating objects (Dunsmore et al. 2002; Dunsmore et al. 2003). Due to the collaborating nature of OO systems, it is important that objects respond as they were intended to under all conditions. It is imperative that the correct method be called in each object and that changes to the object's state be consistent and correct. The aim is that incorrect, missing or error-prone method calls will be discovered by using this technique.

From the system's use cases, the inspector generates multiple scenarios. The inspector is then forced to inspect the class in the context of how it is used and responds within each scenario. Upon completing the inspection, the object's state should match the expected state listed in the scenario. If the states do not match, this suggests that there is a defect within the code. In this manner, the class is inspected in a dynamic way, with the intention of identifying defects through its usage, rather than statically by inspection with a checklist.

To execute a Use-Case Reading inspection, the inspector begins the inspection with a scenario of a system interaction which is described in a sequence diagram. The inspector lists the different state changes that should occur within the system and also what the final states of the object should be. The inspector then traces the execution path through the sequence diagram scenario. Each time the sequence diagram refers to code specifically included within the software inspection, the inspector goes to that code and inspects it, making notes regarding method calls and

changes in object state in order to identify any defects. Once the scenario has been completed, the actual changes in state are compared with the expected changes in state, thereby identifying defects within the system if the states do not match.

A benefit of the UCR technique is that the code is explicitly checked against the requirements described in the scenarios that are generated from the system's use cases. However, a weakness is that certain methods within a class will not be inspected from the created scenarios and these still need to be inspected by the application of a different technique (Dunsmore et al. 2003). This would lead to an increase in the time spent on inspections and therefore an increased cost would be incurred when using this inspection technique.

2.4.5 Usage-Based Reading (UBR)

Usage-based reading is an inspection technique that aims to detect the defects within the system that will cause the largest usage problems. The goal is to find the defects that affect the user experience most, and not to simply find the greatest number of defects. Table 2-2 presents an overview of recent publications regarding the work that has been carried out on the UBR technique. Results indicate this to be an effective means of detecting defects within design documents.

Table 2-2 Usage-Based Reading Studies

Author(s)	Study Details	Results
Thelin et al. (2002)	Investigate basic UBR principle compared with CBR.	UBR is effective in detecting defects that most affect the user.
Thelin et al. (Thelin et al. 2003)	Compared UBR to CBR on design documents.	UBR was more effective in detection than CBR.
Thelin et al. (Thelin et al. 2004)	Investigate the information needed by individual inspectors using UBR	UBR is an efficient method to detect defects.
Winkler et al. (2004)	Replicate Thelin et al. (Thelin et al. 2003).	UBR expert knowledge impacted on results, UBR was more effective than CBR.
Thelin et al. (Thelin et al. 2004)	Replication of Thelin et al. (Thelin et al. 2003).	Results supported earlier work.

Thelin et al. (Thelin et al. 2001; Thelin et al. 2003) derived the technique from the Statistical Usage Inspection (SUI) technique described in the thesis of Olofsson and Wennberg (Olofsson, & Wennberg1996). The aim of the SUI technique was to certify a software product's reliability by testing it in accordance with the system's expected usage.

The core of the Usage-Based Reading technique is the use of prioritised use cases. Prior to inspection, use case scenarios are prioritised, generally by a user of the system or one who knows how the system will be used. The prioritised use cases then function as a tool to focus the inspector's attention on the artefact s/he is inspecting. Design documents as well as code have been inspected using this technique.

The inspector works through the use case scenarios, starting with the highest priority scenario. The inspector systematically traces the scenario through the artefact under inspection ensuring that all needed functionality exists and is correct. In this manner,

the prioritisation is designed to catch those defects that most affect the system's usability.

A strength of this technique is its strong focus on the way in which the system will be used by the client. Ascribing the importance of defect detection to the areas that will be most used leads to a decrease in user dissatisfaction.

A weakness in this method is that, although defects are found in high priority scenarios, a defect in part of the system that is used infrequently may actually cause more damage to the user as its overall importance to the correct functioning of the system is sometimes greater.

2.4.6 Stepwise Abstraction

The stepwise abstraction reading technique is a technique originally recommended for use with poorly documented programs. The technique was aimed at "verification of correctness or the determination of program function" (Linger et al. 1979, p. 148).

One starts at the lowest level within the program, the code, reading it and writing abstracts describing the functionality. Depending upon the documentation that already exists, describing the code determines the way in which the inspector goes about reading the program. There are two options. First: when the program's function is stated without general abstractions describing it, then abstractions are created from the code and then compared with the program's stated functionality to determine correctness. Second: if general abstractions are given, these are used as "anchor points" and the inspector works from these to lower levels, writing the

abstractions for use in determining the program's correctness.

An advantage of this reading technique is similar to that of the Abstraction-Driven Reading technique. Abstractions can be added to the documentation, assisting developers at a later stage with the development or maintenance of the program. However, a disadvantage with using this reading technique is that the time taken to create the abstractions must be factored in to the overall development cost and may be too high in the long run.

2.4.7 Scenario-Based Reading (SBR)

Scenario-Based Reading (SBR) is an inspection technique originally aimed at detecting defects within Software Requirements Specifications (SRS) (Porter, & Votta 1994).

Porter and Votta believed that a more effective technique would be one that gave "individual reviewers specific detection responsibilities and specialised techniques for meeting them" (Porter, & Votta 1994, p. 104). With this technique, scenarios are created that are a series of steps aimed at detecting specific defect types. Each inspector walks through one scenario, the rationale being that with multiple inspectors implementing different scenarios, the artefact under inspection are thoroughly inspected.

The participants in the Porter and Votta (1994) study were graduate students and an improvement of almost 35% by using the Scenario-Based Reading technique over the Ad-hoc or Checklist methods was reported.

Porter and Votta continued to test the Scenario-Based Reading technique by increasing the sample size. The results indicated improvements in defect detection using this technique over Ad-hoc and Checklist-Based Reading, specifically in the areas for which the inspectors were responsible (Porter et al. 1995).

These studies were later replicated using industry professionals (Porter, & Votta 1998). They showed an improvement of defect detection by industry professionals of between 21% and 38% and with students, an improvement of between 35% and 51% was achieved.

The advantage of this reading technique is that well written scenarios not only produce good results; they can be used at a later point by developers and inspectors working on the system. A disadvantage of this technique is that scenarios that are not well written reduce the effectiveness of the reading technique. This means that a large investment of time and effort is needed to create scenarios that lend themselves to effective defect detection, and this time and effort allocation must be taken into account within the project's budget and may prove to be expensive in the long term.

2.4.8 Perspective-Based Reading (PBR)

Perspective-Based Reading (PBR) is aimed at those who will use the specific artefact under inspection. Hence, each inspector is reading the artefact from a different "perspective". "The goal of PBR is to provide operational scenarios where members of a review team read a document from a particular perspective, e.g., tester, developer, user" (Basili et al. 1996, p. 133).

The underlying assumption made by Basili et al. in Perspective-Based Reading is that, by having different inspectors apply different inspection techniques, the inspections are more thorough and comprehensive than if all the inspectors had applied the same inspection technique. Consequently, the artefact under inspection is more comprehensively examined and a wider range of defects may be detected.

Regnell et al. (2000) further tested the Perspective-Based Reading to ascertain whether the different perspectives detect different defects and whether or not the different perspectives have different efficiency levels. Their results indicate that the different perspectives neither detected significantly different defects, nor were more efficient than one another.

However, the strength associated with the different perspectives taken by Basili et al. (1996) regarding the different types of defects to be detected, was not supported by Regnell et al. (2000).

2.4.9 *N*-Fold Inspections

The *N*-Fold inspection technique was first introduced by Martin and Tsai (1990). The inspection technique is a formal technique that focuses on detecting defects within the User Requirements Document (URD). To implement this technique, a formal inspection of the artefact is conducted using a checklist. The technique differs from the standard Checklist-Based Reading inspection as there are *N* independent teams carrying out the same inspection using the same checklist upon the same artefact. Martin and Tsai suggest this technique be implemented in the early phase of

the development of mission-critical software. The underlying assumption is that by having multiple (*N*) different teams inspecting the same artefact, defects will be detected that may not have been detected had only a single team carried out the inspection.

One of the strengths of the *N*-Fold inspection technique is the high number of people who examine the same document. With more people inspecting the document for defects, it is a natural assumption to expect more defects to be detected within that artefact. The weakness however, is that multiple teams inspect the same artefact, during which time they could be inspecting other artefacts within the system.

2.4.10 Phased Inspections

The phased inspection process is one that arose from Knight and Myers' (1993) strong belief in the crucial role inspections should play in the entire software development life cycle. The phased inspection's over-arching aim is that every artefact be inspected at each stage of the development process.

There are four goals that underpin the phased inspection. The inspection must be:

1. rigorously applied with the results specific to an artefact and repeatable;
2. a customisable process so it becomes more than a defect detection technique;
3. computer-supported thereby reducing human resources needed; and
4. efficient, maximising resource usage.

The process involves a set of co-ordinated limited inspections called 'phases'. During the inspection, the artefact is inspected with respect to a single or small

group of traits. The trait or group of traits must be small enough to ensure it is a manageable size for the inspector. Besides examining an artefact for defects, the phased inspection can also be used to check that the artefact under inspection complies with an organisation's design rules, e.g. the source code complies with the organisation's coding standard.

The inspectors execute the inspection task by going through a series of questions from a checklist. In this regard, the phased inspection mirrors a checklist-based reading inspection that is applied to each and every software artefact. It checks those artefacts for specific defects as well as compliance with other design standards within the organisation.

A strength of Phased Inspections is that each and every stage of the development process is inspected for correctness along with each created artefact. This indicates a rigorous inspection of the system and its supporting documentation has been carried out. A weakness with this inspection technique is that, as several different aspects of the artefact are often checked by the inspection, it is possible that the inspector reports defects that are not crucial to the system performing correctly, but rather defects that have very little impact on performance.

2.4.11 Traceability-Based Reading (TBR)

Traceability-Based Reading was developed by Travassos et al. (1999) specifically for inspecting high-level Object-Oriented design documents created using the Unified Modelling Language (UML). This technique looks specifically at the high-level design documents such as class diagrams, sequence and collaboration

diagrams, state machine diagrams and package diagrams.

The strength of an inspection method such as this one is that it ensures that the UML representation of the system is correct. This means that developers who use this diagram to create the system, have a correct representation and can begin to implement it. A weakness of this inspection method is that all too often these types of documents do not get updated when requirements change. This may mean that although the original document was correct, it is no longer up to date. Therefore, it is an incorrect system representation. All UML diagrams must be kept up to date for this inspection technique to be effective.

2.5 Program Comprehension

It is estimated that between 60% and 80% of a software system's life time is spent with developers attempting to maintain it (Bennet 1990; Fjelstad, & Hamlen 1979; Pfleeger Lawrence, & Atlee 2010; Swanson, & Beath 1989). Program comprehension and understanding forms a major role in this software maintenance and evolution phase (Mayrhauser, & Vans 1993; Linos et al. 1993; Storey et al. 1997). Hence, for the past 30 years the maintenance and evolution of software systems have been and remain both time consuming and expensive processes.

In maintaining and evolving these software systems, developers spend a large portion of this time attempting to understand the system in order to carry out the needed maintenance. It has been reported that between 50% and 90% of this time is invested in developer comprehension (Canfora, Mancini, & Tortorella 1996; Rajlich 1994; De Lucia, Fasoline, & Munro 1996; Hartzman, & Austin 1993; Dunsmore,

Roper, & Wood 2000). Developers' lack of correct understanding of a software system is listed as a major reason for software maintenance failure.

O'Brien et al. (2004) estimate that up to 35% of the total cost of software systems is invested simply in ensuring developer comprehension. Pressman (2005) also highlights that, due to the transient nature of software developers, there is a high probability that nobody from the original development team is still working on the software when new members join the team. Hence, because there is no organisational or project memory, new team members must work through the code in order to understand what the system does and how it operates.

One area that has been examined is in understanding developers' cognitive processes as they build their understanding of the software system they are working on. Several models have been developed that attempt to explain the manner in which software developers acquire their comprehension of a software system's operational and functional behaviours (Brooks 1983; Littman et al. 1987; Mayrhauser, & Vans 1995; Robson et al. 1991; Shneiderman 1977).

The two general cognitive models used to explain developer comprehension usually highlight either the programmer's acquisition of knowledge by studying the function of the program (the tasks it performs), or how the program performs its designated tasks (control flow) (Mayrhauser, & Vans 1993). As a software developer embarks on maintaining and evolving a system, it is crucial that s/he understands both the functional and control flow of the system. Without this understanding, the developer may make changes that appear correct in one location but wreak havoc in the system

in another location.

Kelly and Buckley (2006) point out that two distinct tracks have evolved in software comprehension research. The first track is how software is represented and what impact these representations have upon a developer's comprehension level. Kelly and Buckley show that these representations are split into two categories: one is external representations, such as system source code and documentation while the other is internal representations such as programming plans (Soloway, & Ehrlich 1989) and beacons (Wiedenbeck 1986). The second track is in the cognitive processes domain developers implement as they go about attempting to comprehend the systems they are working on.

Figure 2-2 shows the software development domain of moving something from the problem domain to the programming domain. The software developer is presented with something that lies in the problem domain. The eventual solution, which will be a piece of software, lies in the programming domain. In moving from the problem to the programming domain, the software developer will move through several different domains until the programming domain is reached with a working software solution. The art of comprehending software is found in recreating the knowledge of these domains and how they interact (Brooks 1983).

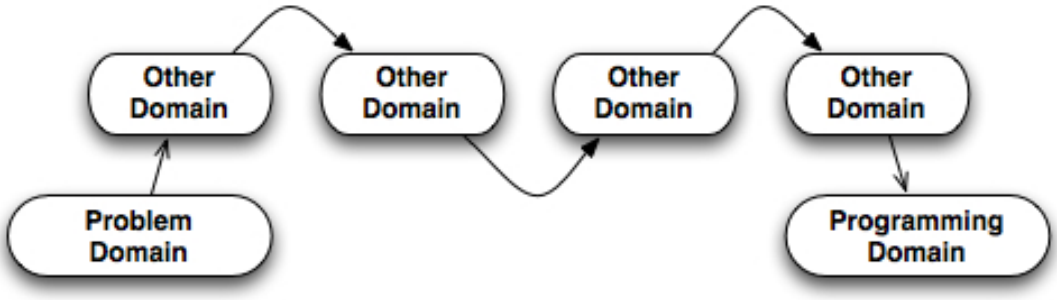


Figure 2-2 Software development, moving from problem to programming domain.

Techniques for software comprehension vary from paper to paper but a common theme is that there is a general lack of documentation to assist the developer. Software documentation quality is sometimes poor or lacking altogether (Briand 2003). This is because documentation is perceived to be time consuming and expensive, with many software development projects not having budgets that allow for accurate and up-to-date documentation (Forward, & Lethbridge 2002). It is therefore of very little use to the new developer attempting to understand the system. There can be long-term ramifications from this lack of or inaccurate documentation. When maintenance needs to be performed, and the original developers are not available, the documentation provides little, if any, assistance to the new developers.

Table 2-3 shows five different cognitive process models that have been described in the literature. A brief description is given within the table on each of the models and a more expansive description of each of the models is given shortly.

Following is a description and discussion of the different cognitive process models describing the processes used by developers as they attempt to understand a system in order to carry out their assigned tasks upon it.

2.5.1 Top Down

Brooks (1983) describes the top-down cognitive model associated with computer program comprehension. This model is a hypothesis-based model. The software developer builds an understanding of the system by creating hypotheses about the program, what it does and how it goes about carrying out that task. The developer then searches through the code to verify the hypothesis as true or false and then continues to refine each of the hypotheses.

In the top-down model, the software developer creates the first general hypothesis as soon as s/he receives any information about the software system. For example, when they are told the name of the program they may then generate their first hypothesis about what the program does. However, if developers have no background knowledge within that domain, then their first hypothesis is usually not generated until they have either read a reasonable amount of information describing the system or used the system themselves.

Table 2-3 Cognitive process models.

Model	Author	Description
Bottom-up	Shneiderman (1977)	"chunks" of code are created and grouped together until the problem has a solution.
Top-down	Brooks (1983)	a global hypothesis describing the whole program is created. Further hypothesis refinement occurs and is tested until the program, in its entirety, is understood (this methodology is also described by Polya (1957) and Wickelgren (1974)described this problem solving methodology)
Systematic	Littman et al. (1987)	a developer systematically reads through the software, building their understanding by looking at data and control flow.
As needed	Littman et al. (1987)	as the name indicates, a developer looks only at what is needed for the task that needs immediate attention.
Intergrated	Mayrhauser & Vans (Mayrhauser, & Vans 1995)	here a developer uses both the top-down and bottom-up methodologies to best assist them in their understanding of the software

Verifying the general hypothesis requires the developer to search the code or documentation to support the hypothesis. The developer starts with a general hypothesis that may describe the global nature of the program. The developer then breaks the hypothesis down into smaller and smaller hypotheses. Each subsequent hypothesis is broken down until a line or several lines of code or supporting documentation verify the hypothesis. In this manner, the developer acquires an understanding of the system from the top, the higher level abstractions, down to the lower level elements, the code and its supporting documentation.

Figure 2-3 diagrammatically represents this process as the developer starts at the top with a general understanding/hypothesis and this hypothesis is reduced in size until it

is verified by code or supporting documentation.

2.5.2 Bottom-Up

The bottom-up program comprehension cognitive process model was described by Shneiderman (Shneiderman 1977; Shneiderman, & Mayer 1979). The bottom-up cognitive model establishes two different knowledge structures: syntactic and semantic.

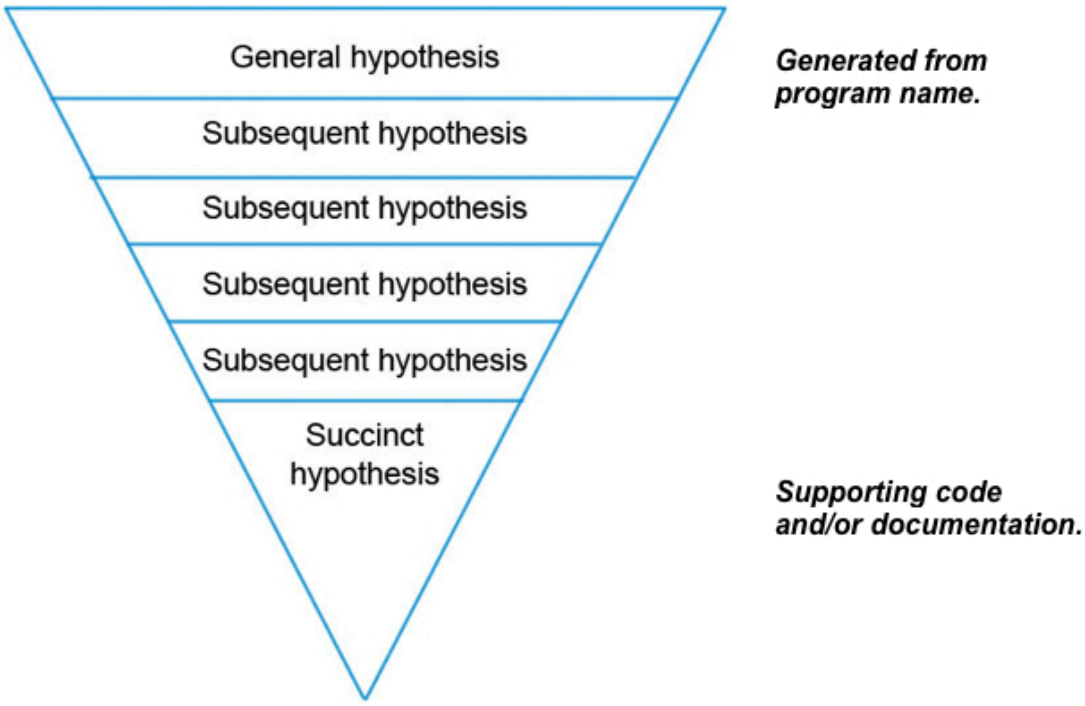


Figure 2-3 The Top-down cognitive model.

The syntactic knowledge is one that is based around programming languages such as understanding how to write a class in Java or how to write a class in C++. The syntactic knowledge is language-dependent.

Semantic knowledge is language-independent. Semantic knowledge arises from consequential learning and is arranged from low-level constructs to intermediate

program structures through to high level problem domain issues. The way in which a developer goes about acquiring this semantic knowledge related to a specific system is known as the ‘bottom-up cognitive model’.

Software developers establish their semantic knowledge through the process of “chunking.” George Miller (Miller 1956) first described the chunking concept. To the software developer, this equates to grouping statements of code together and understanding them as a chunk rather than studying the system one line of code at a time. The developer then groups together more lines of code to form another chunk, and understands that chunk. The chunks themselves are then “chunked” together forming a larger chunk enabling the developer to understand system functionality through these grouped chunks.

Figure 2-4 demonstrates the bottom-up cognitive model. At the bottom of the diagram are individual rectangles representing small chunks. At the lowest level, each chunk represents a small amount of grouped code that identifies a small piece of functionality. Moving up one level, each rectangle represents increased functionality, which is understood by the developer who has first acquired an understanding of the lower-level chunks. This process continues until the entire program, or a particular section of the program, has been understood by the software developer.

This cognitive model works well alongside procedural programs. With procedurally structured programs, it is possible to read a program starting at the main function or procedure and follow each call through the program until execution has been

completed.

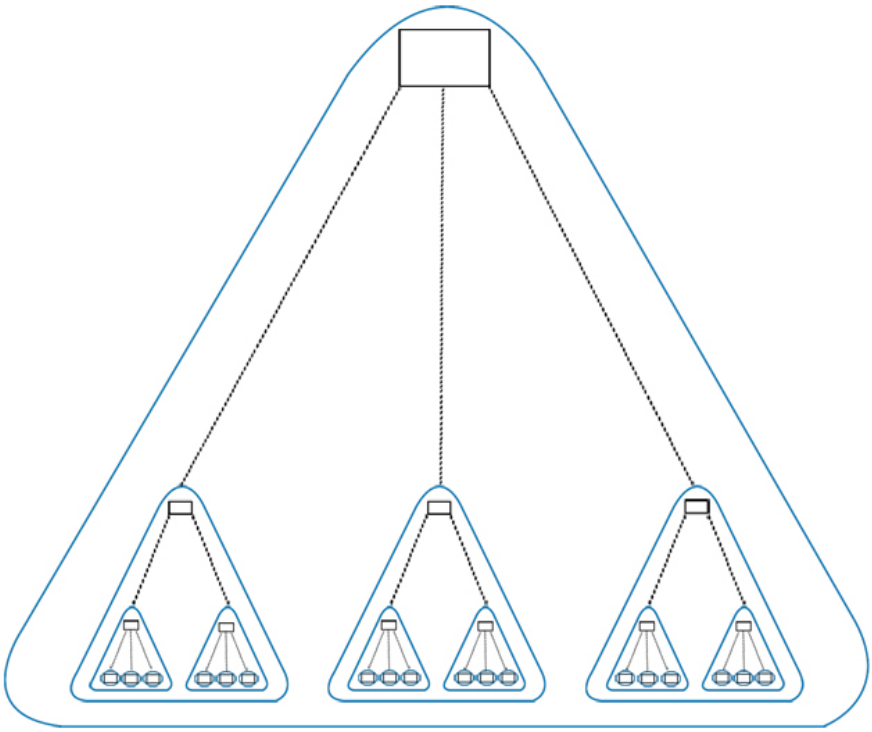


Figure 2-4. The Bottom-up cognitive process model.

However, with the object oriented programming paradigm, reading a program from the start of the main function or procedure and following each call through the program is more difficult due to the nature of object oriented programming: polymorphism, inheritance, late binding, small methods etc. These object oriented features now pose a challenge to developing a system-wide understanding using the bottom-up cognitive model.

2.5.3 As Needed

The As Needed model is a mental model, describing how developers attempt to understand a software system, described by Littman et al. (1986; 1987).

In the As Needed mental model, developers aim to understand the code that is local to where the maintenance will occur. The developer's main concern is to understand the behaviour immediately surrounding the code that will be modified. Generally, an attempt to understand the entire system is not made when the As Needed model is used.

In attempting to localise the area to be understood, often during the maintenance and/or modifying the code, developers will need to gather more information and understanding as to what the system is doing and how it is doing it as they encounter code that they have not learned or understood due to it being outside the local area. It is while they perform the modification that questions arise determining what they need to learn and understand.

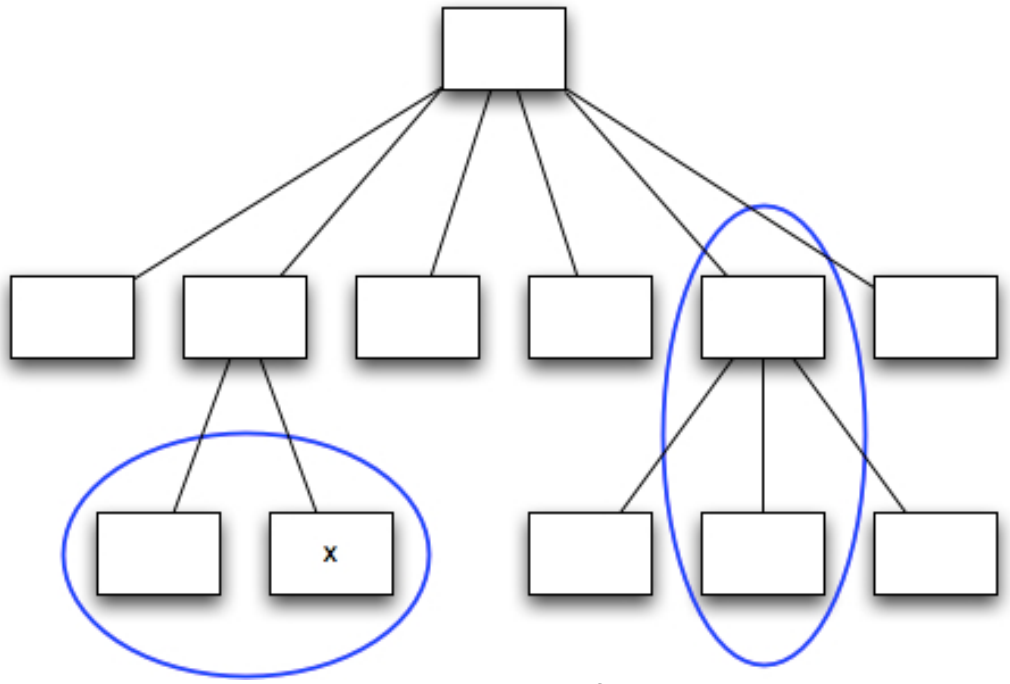


Figure 2-5 The as needed cognitive model for program understanding.

Figure 2-5 shows a diagrammatic representation of a software system. The module labelled ‘x’ requires a modification to be made to it. The developer is attempting to understand the program in a localised manner, not the entire system. They attempt to understand only what it is necessary for them to know in order to make the modification. The blue ellipse on the right hand side indicates the area that they have already looked at for the required change. However, the change was not in that location. The required change is to the module labelled ‘x’ and now the developer has understood that which is encapsulated within the blue circles. In this manner, the understanding is kept local.

A weakness of the As Needed model is that a change in module ‘x’ may actually have an effect somewhere else within the system and by containing the understanding to the local code area, the developer may not be aware of the implications that a change will have upon the wider system. Littman et al. (1986; 1987), note that the developers who applied the systematic model gained a greater understanding of the relationships within the system than did those who used the As Needed strategy.

2.5.4 Systematic

The systematic cognitive model also described by Littman et al. (1986; 1987) is a mental model used by developers as they attempt to understand a software program during maintenance.

The systematic model is one in which developers attempt to understand the program’s behaviour in full, before they attempt to make code modifications. In

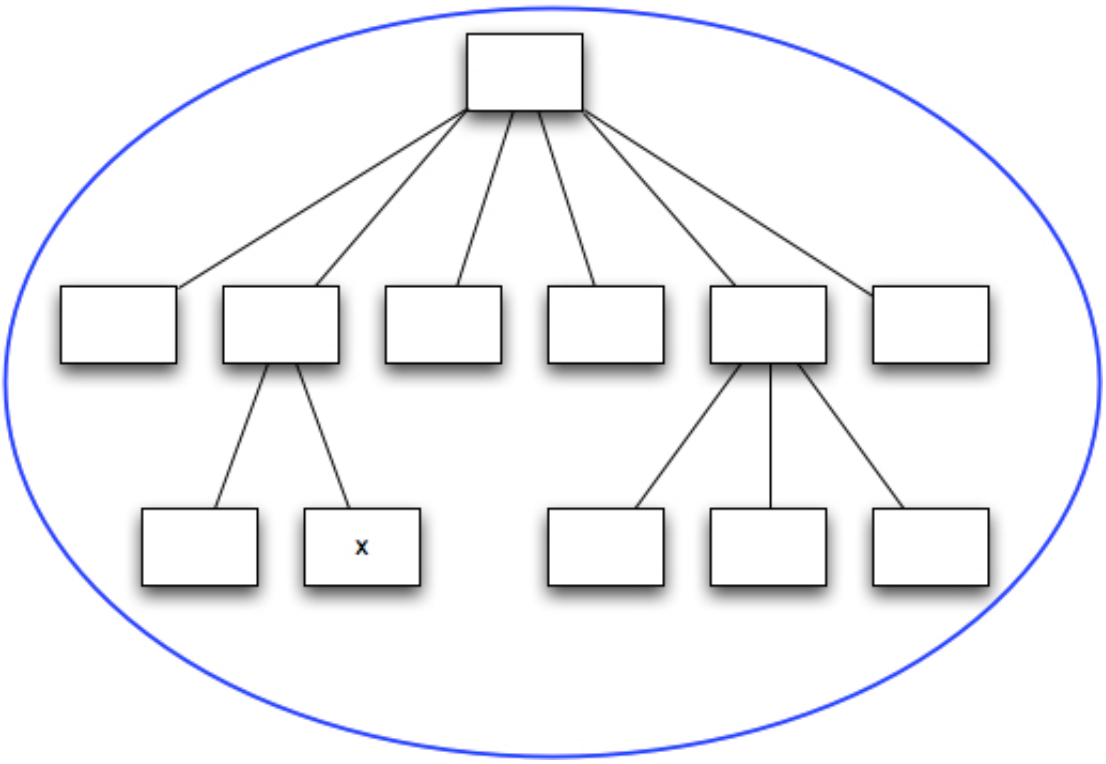


Figure 2-6 The systematic cognitive model for program understanding.

order to do this, they trace both data and control flow through the program. They walk through the execution of data flow paths between modules in order to understand component interactions.

Figure 2-6 represents the same software system as shown in Figure 2-5, with the same change requirement. The module labelled ‘x’ requires a modification to be made to it. When applying the systematic cognitive model, the developer attempts to understand the entire system, shown in the diagram by the all encompassing blue circle. Once the developer understands the system, s/he makes the changes.

When using the systematic model, developers attempt to gain an overall understanding of what the software does and how the software goes about doing this. They do this so that when they make the changes needed to the code, they are able to account for the interactions within the code so that their changes do not cause these

interactions to fail i.e. to prevent the introduction of new bugs.

The challenge with the systematic cognitive model is that with the expanding size of software systems, it becomes very difficult to fully understand or to have a complete overview of that everything a system is doing.

2.5.5 Integrated

The integrated comprehension model was described by Mayrhauser and Vans (1993; 1995). The model evolved as they observed professional developers performing maintenance on large systems. The developers did not exclusively use a bottom-up model, nor did they exclusively use a top-down model; instead the developers alternated between both the bottom-up and top-down models.

These observations were consistent with those of Rist (1986), who observed that as programs become more complex, developers interchangeably use both the top-down and bottom-up learning methods rather than using one model exclusively.

The integrated comprehension model is made up of four elements:

1. Program model;
2. Top-down;
3. Situation model; and
4. Knowledge structure.

Elements 1 – 3 are comprehension processes, while element four is a requirement necessary for elements 1 – 3 to occur. The developer's internal representation, also

called a mental model, is captured through the processes of elements 1 – 3. The knowledge base, also known as the knowledge structure, supports the processes with the needed and related information to allow the comprehension to occur (Mayrhauser, & Vans 1993; Mayrhauser, & Vans 1995). Also, new information that arises through elements 1 – 3 is placed into the knowledge base.

In describing the programming model within the Integrated model, Mayrhauser and Vans draw on the work of Pennington (1987a; 1987b) where she noted that the control flow (program model) is the first mental abstraction of the system that developers build, especially when the code is unfamiliar to them. They then take Vessey's (1985) debugging model tasks, mapping them into the program model. Vessey's (1985) tasks included:

- Reading comments and/or other documentation.
- Micro-structure examination: where the developer examines modules in text sequential order.
- Macro-structure examination: Web developer examines modules in control flow sequential order.
- Data structure examination.
- Data slicing: following and examining changes to variables as the program executes.
- Chunking: taking what has been learned and placing it into their knowledge structure.

Mayrhauser and Vans (1993) continued by mapping the tasks they identified within their integrated model into the top-down model. These tasks included establishing a

high-level overview, deciding which code segment to next examine, ascertaining the code segments apropos to their current hypothesis and the generating and checking of hypotheses. The developers would move to the situation model if they confirmed one of their hypotheses.

The situation model is created in one of two ways: functional knowledge obtained during understanding is mapped to higher level plans, or code abstractions are mapped in a bottom-up manner (Mayrhauser, & Vans 1993).

The knowledge structure or knowledge base is also considered the developer's long-term memory. Here in long-term memory the developer organises, groups and partitions the information acquired during the comprehension process.

The integrated model is quite complex as it incorporates several of the other comprehension models discussed earlier. Of all the models discussed in this section, the systematic application of this comprehension process appears to be the most holistic approach to program comprehension.

2.6 Reverse Engineering and Visualisation

Reverse engineering is an engineering process that is often associated with hardware manufacturers. Hardware manufacturers disassemble a competitor's product in an attempt to understand how it does what it does (Pressman 2005). Sommerville (2007), and Pfleeger and Atlee (2010) describe reverse engineering in a software context as the process that extracts information about the specification and design from the source code.

With many software systems, especially legacy systems, the source code is the only documentation that actually exists and describes the system. Hence, when developers need to understand the system, they reverse engineer it in order to help them understand what the system is doing and how it is doing it.

The reverse engineering process is widely discussed within the literature, for example, the Working Conference on Reverse Engineering. Reverse engineering falls outside the scope of this thesis and therefore will not be discussed in greater detail.

Program visualisation is the creation of 2-D and/or 3-D visual representations of software code or data in either a static or dynamic form (Stasko 1998). The tools used to create the visualisations generally use metrics extracted from the reverse engineering process. Figure 2-7 shows a piece of software visualised. The program visualisation goal assists the developer to understand the code and increases the software production efficacy (Diehl 2007).

Code visualisation falls outside the scope of this thesis and hence will not be discussed in any further detail.

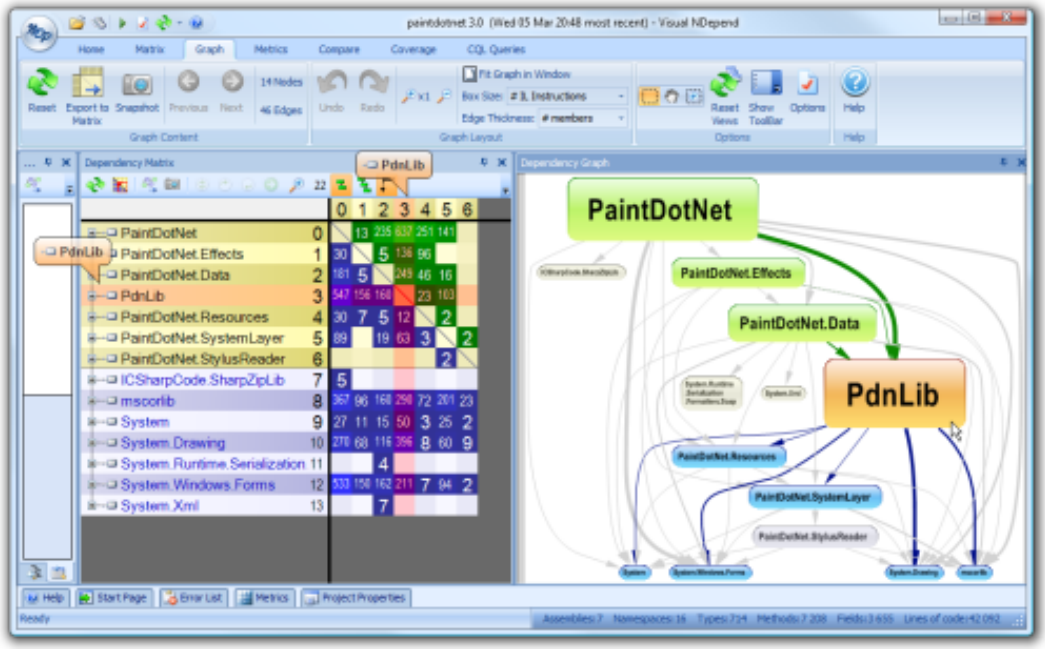


Figure 2-7 A code visualisation.[2]

2.7 Delocalisation

A major challenge faced by OO code inspectors is delocalisation. Soloway et al. (Soloway et al. 1988) described delocalisation as the phenomena where related code is spread throughout many different locations of the code base.

The OO programming paradigm introduced many features not commonly available in procedural programming, such as function and operator overloading, inheritance, dynamic (late) binding and polymorphism. It was expected that these features would improve software quality with better data abstraction and information hiding, more readily allowing for concurrency and adaptability to changes in the real world. However, these additional features increased program complexity, causing other complexities within these systems (Booch 1986; Lejter et al. 1992; Wilde, & Huitt 1992; Wilde et al. 1993).

[2] The image is in the public domain and was downloaded from Wikipedia.

OO programs cannot simply be read from top to bottom. An OO program written in a delocalised manner becomes difficult to understand since the inspection of a single class may rely on several other classes for it to be fully understood. The code relied upon by the class under inspection may be contained in another class. If the class containing this code is not within the scope of the inspection, inspectors make assumptions about it which may be either correct or incorrect. Accurate assumptions are difficult to make when inspecting code in delocalised components. This problem is not unique to OO programs. Programs written using a procedural technique also face similar issues with function calls. However, it appears that the OO paradigm greatly increases the delocalisation problem (Dunsmore et al. 2003).

2.8 Bloom's Taxonomy

A group of educators, lead by Benjamin Bloom, developed a classification taxonomy that identified six different cognitive levels in the learning process. The taxonomy itself is now known as Bloom's Taxonomy. The book they published was titled *The Taxonomy of Educational Objectives, The Classification of Educational Goals, Handbook I: Cognitive Domain* (1956) and it has been called "…one of the most influential educational monographs of the past half century" (1994).

Bloom's taxonomy is a classification taxonomy that identifies different cognition levels potentially exhibited during learning. Within the taxonomy, the six different levels range from low through to high cognitive levels.

The taxonomy has been widely embraced and used within educational disciplines

and in many cases is still the standard taxonomy used when testing and evaluating subjects, developing curriculums, and within teacher education (1994).

The six levels are knowledge, comprehension, application, analysis, synthesis/creation and evaluation. This was the order originally established by the educators. However, within the education field there is much debate and discussion as to whether or not synthesis/creation and evaluation should change places.

In a revision of the taxonomy carried out by Anderson et al. (2001) they did actually reverse the order of synthesis/creation and evaluation levels as well as make slight modifications to the other categories. The two alternative orderings and naming are shown in Table 2-4.

Table 2-4 Bloom's Taxonomy categories, the original and revised versions.

Original by Bloom (1956)	Revised by Anderson et al. (2001)
Knowledge	Remember
Comprehension	Understand
Application	Apply
Analysis	Analyse
Synthesis/Creation	Evaluate
Evaluation	Create

Each category, cited from Bloom (1956), is listed and briefly described below, with an example of how each might translate to a programmer's context:

1. **Knowledge**: is the process of remembering. It is the recalling or remembering of facts, specifics and patterns. In programming, this may be demonstrated by the recalling of a while loop condition.
2. **Comprehension**: is considered as the lowest level of understanding within the taxonomy. This is where one understands what was communicated and

can explain it to another in a different manner. For the programmer, this may be indicated by his/her ability to describe the function of a code fragment.

3. **Application**: is the use of learned knowledge in a different environment. The application here is in solving a problem. The programmer makes a change to the code which leads to the code being more efficient.
4. **Analysis**: is where complex information is recognised and hidden meanings are revealed. For example, a programmer demonstrates analysis level cognition understanding by describing how a field or method operates in the context of the wider system and its role in object collaboration.
5. **Synthesis**: also referred to as creation, is where a new whole is formed by combining different elements and parts. For example, a programmer creating a new method that adds new functionality, not just modifying existing functionality in the code, would be operating at the synthesis level.
6. **Evaluation**: is where judgements regarding the work etc. are made. Here, programmers may give an appraisal of the code, evaluating whether it is appropriate for its intended purpose.

2.8.1 Bloom's Taxonomy in Software Engineering Studies

Bloom's Taxonomy has been implemented in software engineering and computer science studies. It has been used for two different applications within these disciplines. One is in the education and training area including overall curriculum design for undergraduate and graduate degrees. In education and training, it is also used for creating courses/units within these degree programs and determining how the objectives for each section of each course/unit is created and evaluated. Table 2-5 provides a sample listing of publications explicitly using Bloom's taxonomy for

the creation and/or evaluation of what is taught within degree courses/units within computing related tertiary programs.

Table 2-5 A sample listing of publications using Bloom's Taxonomy in SE/CS education.

Author	How Bloom's Taxonomy was applied/used
Lister (Lister 2000)	First year programming approach addressing all six stages.
Lister & Leaney (Lister, & Leaney 2003)	Used for creating a reasoned, straightforward marking philosophy.
SWEBOK (Software Engineering Body Of Knowledge) (2004)	Used to specify the expected understanding levels of each topic within its Knowledge Areas (KA) for graduates.
Oliver et al. (Oliver et al. 2004)	Analysed cognitive difficulty of six courses and scored them with a Bloom's ranking.
Whalley et al. (Whalley et al. 2006)	Tested sections of program comprehension categorised using Bloom's taxonomy.
Khairuddin & Hashim (Khairuddin, & Hashim 2008)	Software engineering assessment using Bloom's Taxonomy.
Thompson et al. (Thompson et al. 2008)	A valuable tool assisting with analysis of programming assessments when consistently interpreted.

The second implemented usage is as a means by which developers' cognitive levels can be measured and assessed during different software engineering development tasks (Buckley, & Exton 2003). Table 2-6 provides a sample listing of publications that explicitly used Bloom's Taxonomy in this manner.

These two different usages are not mutually exclusive. An understanding of the cognitive levels at which developers are operating when they are performing different software related tasks allows for the creation of courses/units that are relevant to these cognitive levels. Tasks, assignments, evaluations etc. can be created to assist students to function at the higher cognitive levels or to encourage students to move towards the higher cognitive levels.

Table 2-6 A sample listing of publications using Bloom's Taxonomy to categorise and measure developers' different cognitive levels.

Author	Description
Xu & Rajlich (Xu, & Rajlich 2005)	Examined cognitive activities while developing and evolving software.
Xu & Rajlich (Xu et al. 2005)	Examines programmer learning during incremental program development.
Kelly & Buckley (Kelly, & Buckley 2006)	Developing a framework in order to compare software comprehension studies.
Kelly & Buckley (Kelly, & Buckley 2009)	Examined if professional programmers operate at all levels of the taxonomy while performing maintenance.

Xu and Rajlich (Xu, & Rajlich 2004; Xu et al. 2005; Xu, & Rajlich 2005) used Bloom's taxonomy to characterise programmer cognitive levels while the latter executed software engineering tasks. The data (think-aloud data, explained shortly) was characterised using a verb table they created that listed different verbs according to the appropriate Bloom's Taxonomy level, to measure the developers' different cognitive levels. That data was analysed for the different verbs, and according to which verbs were in the data, it was categorised into the corresponding taxonomy level. Table 2-7 is a reproduction of the verb table from Xu and Rajlich (Xu, & Rajlich 2005, p. 402). They noted that with their approach, the higher cognitive levels of the taxonomy were not adequately measured.

Kelly and Buckley (Kelly, & Buckley 2006) point out that although Xu and Rajlich's method was both simple and efficient, there were two problems with it: the first was that ambiguity is found in that some verbs appear in more than one taxonomy level and the second is a reductionist problem in that some data is not catered for and because of its omission, valuable information may be lost. Hence, Kelly and Buckley (Kelly, & Buckley 2006) created a Context-Aware Schema for Bloom's Taxonomy.

Table 2-7 The six Bloom's levels and corresponding verbs. Reproduced from (Xu, & Rajlich 2005, p. 402).

Bloom's Levels	Sample Verbs
Recognition	collect, copy, define, describe, enumerate, examine, identify, label, list, name, quote, read, recall, retell, record, repeat, reproduce, select, state, tell
Comprehension	associate, cite, compare, contrast, convert, differentiate, discuss, distinguish, elaborate, estimate, explain, extend, generalize, give, group, illustrate, interact, interpret, observe, order, paraphrase, review, restate, rewrite, subtract, trace
Application	administer, apply, calculate, capture, change, classify, complete, compute, construct, demonstrate, derive, determine, discover, draw, establish, experiment, illustrate, investigate, manipulate, modify, operate, practice, prepare, process, produce, protect, relate, report, show, simulate, solve, use
Analysis	analyse, arrange, breakdown, classify, compare, connect, contrast, correlate, detect, diagram, discriminate, distinguish, divide, explain, identify, illustrate, infer, layout, outline, points out, prioritize, select, separate, subdivide
Synthesis	adapt, combine, compile, compose, construct, correspond, create, depict, design, devise, express, format, formulate, facilitate, improve, integrate, invent, plan, propose, rearrange, reconstruct, refer, relate, reorganize, revise, specify, speculate, substitute
Evaluation	appraise, assess, conclude, criticize, convince, decide, defend, discriminate, evaluate, explain, grade, judge, justify, measure, rank, recommend, reframe, support, test, validate, verify

2.8.2 Context-Aware Analysis Schema Using Bloom's Taxonomy

Kelly and Buckley (Kelly, & Buckley 2006) proposed a context analysis aware schema using Bloom's taxonomy to categorise developers' cognitive processes demonstrated while performing different software maintenance tasks. The schema requires developers to "think-aloud" as they perform the different tasks required of them.

The Context-Aware Analysis Schema is based on sentence or utterance analysis. The think-aloud data is recorded, transcribed and broken down into sentences or utterances. Each sentence or utterance is then categorised into one of the six levels of Bloom's taxonomy to identify the cognitive level at which the developer was operating when s/he made that utterance. Each utterance is categorised according to both its content and the previous two utterances. This enables the utterance to be categorised within its applied context.

The Context-Aware Analysis Schema (Kelly, & Buckley 2006) accounts for only five of the six categories within Bloom's taxonomy. The reason for this is that their original proposal was made in the context of software maintenance. Consequently, the Synthesis category was omitted because it required the creation of something new, such as adding new functionality or building an entirely new program. Within their proposal's context, software maintenance, the building of something new does not occur, hence the rationale behind its omission.

The Context-Aware Analysis Schema's strength is that utterances or sentences are categorised contextually and not simply syntactically. The classification into the different taxonomy levels is dependent on the context of the utterance and not simply what verb is used. A weakness of the schema is that there is no category for the Synthesis/Creation level in Bloom's Taxonomy. The second weakness is that utterances are categorised according to the context of the two previous utterances, consideration is not given to the context of the utterances that follow.

2.9 Think-aloud Data Collection

Think-aloud or Protocol Analysis is a process that evolved from within the Psychology discipline. In attempting to gain an understanding of one's cognitive processes, psychologists began looking at the eye movements, physical movements and verbalisations of subjects' thoughts as they carried out different given tasks (Ericsson, & Simon 1993). The verbalisation of the subjects' thoughts has been called think-aloud or protocol analysis (referred to as think-aloud from this point forward). Eye and physical movements are beyond the scope of this thesis but the think-aloud data collection technique has been utilised within this thesis.

Think-aloud is a process whereby participants verbalise their thoughts and actions while performing a task (Ericsson, & Simon 1993). The verbalisations are recorded as data collected from the task performed and are then analysed in order for conclusions to be drawn from them.

The think-aloud data collection method has been well used in studies reported within the Software Engineering and Computer Science literature, examining participants' cognitive levels expressed while carrying out the given task. Table 2-8 is a sample of studies that have been conducted using the think-aloud data collection method. It is not exhaustive but indicates nevertheless that this data collection method has been used widely within the Software Engineering and Computer Science empirical research domain.

The think-aloud data collection method has assisted in the understanding of the ways in which software developers work and the cognitive processes they use as they

perform different tasks on software systems. From studying the articles listed in Table 2-8, a vast amount of what is actually understood about cognitive processes developers apply when performing different tasks is due to think-aloud data collection and analysis.

A disadvantage of the think-aloud data collection and analysis method is that it is labour intensive. Once the recordings have been completed, they are transcribed in order for categorisation and analysis. This transcribing process consumes vast amounts of time if the researchers carry it out themselves or is very expensive if carried out by professional transcribers. Due to the labour intensiveness and/or expense involved in using this method, in many cases small sample sizes are used with the studies reported within the literature.

2.10 Participants: Students or Industry, Prior Studies

Empirical software engineering research aims to improve software quality by developing and improving design and development processes and methodologies, inspection and maintenance techniques as well as tool support needed for these processes (Sjøberg et al. 2005).

Table 2-8 A sample of Software Engineering/Computer Science research papers where Protocol Analysis has been used.

Author(s) & Year	Reported results summary
Anderson et al. (1984)	Programmer behaviour is strongly influenced by structural analogy. Solution structure is usually hierarchical.
Adelson & Soloway (1985)	Developers: create mental models, simulate to include familiar material, systematically expand, and implicitly constrain designs in unfamiliar domains.
Vessey (Vessey 1985)	Investigated debugging processes of experts and novices and contributed to programming expertise theory.
Letovsky (1986)	Derived a computational model of the mental processes of programmers.
Littman et al. (1987)	Identified two different processes used by developers in program understanding: Systematic and As-needed.
Fisher (Fisher 1987)	Shows a multi-purpose computer aided protocol analysis system.
Pennington (Pennington 1987a)	Professional developers who work at high comprehension levels think about both the program they are developing and the world in which it will be deployed.
Bergantz & Hassell (Bergantz, & Hassell 1991)	Developing psychological complexity metrics from program comprehension models. Identified key informational relationships for programmer comprehension of PROLOG.
Koenemann & Robertson (Koenemann, & Robertson 1991)	The comprehension activities of expert programmers modifying a PASCAL program were studied which was a goal oriented, hypotheses driven process.
Detienne (Detienne 1991)	Analysed the characteristics of experienced programmers cognitive mechanisms in code reuse activities.
von Mayrhauser & Vans (Mayrhauser, & Vans 1993)	Describes the integrated cognitive model approach, the combination of the top-down and bottom-up cognitive models.
Pennington et al. (Pennington et al. 1995)	Analysed different cognitive activities and strategies of developers as they went through a design process.
von Mayrhauser & Vans (Mayrhauser, & Vans 1996)	Examined the different understandings that occur between maintenance tasks on small and large scale software systems.
Xu et al. (Xu et al. 2005)	Examined programmer learning in incremental development examining self-directed learning theory.
Xu & Rajlich (Xu, & Rajlich 2005)	Examining a variation to protocol analysis, dialog-based protocol, attempting to lessen the Hawthorne and placebo effects.
Kelly & Buckley (Kelly, & Buckley 2006)	Created a context aware schema to classify think-aloud data into Bloom's Taxonomy categories (excluding Synthesis).
Kelly & Buckley (Kelly, & Buckley 2009)	Professional programmers tend to operate at all levels in Bloom's taxonomy while maintaining code.

Sjøberg et al.'s analysis of the literature showed that studies in the Software Engineering (SE), Computer Science (CS), Information Technology (IT) and Information Systems (IS) literature have often used students as participants. Several also used industry professionals. In a literature survey of 12 software engineering journals and conferences over a ten-year period (1993 to 2002), 103 of the 5453 articles described studies in which either individuals or teams performed some software engineering task. Student participants were involved in 81% of the studies while professional developers were involved in 24% of the studies (Sjøberg et al. 2005).

Table 2-9 Listing of articles discussing participants within empirical research.

Authors	Summary of comments
Weinberg (1971)	Using trainees for studies gives results about trainees.
Curtis (1980)	Generalizations about professionals should be careful when novices are used.
Curtis (1986)	Results about novices should produce better teaching methods and tools.
Potts (1993)	Research occurs, results derived and then industry should be involved in the application of the results.
Glass (1994)	Software engineering research has been done independent of industry.
Votta (1994)	Highlights that recommendations from Curtis (1986) haven't been implemented.
Höst et al. (Höst et al. 2000)	Compared student and professional developers, concluding differences are only minor and under certain conditions students can be used and not professionals.
Sjøberg (2002)	Studies should reflect industrial setting.
Sjøberg (Sjøberg et al. 2005)	The lack of reality in studies can cause technology transfer from academia to industry to be very slow.

Table 2-9 lists a sample of studies reported within the literature that have specifically discussed the participant experience breakdown contained within empirical research. The comments' summary in the table indicates that varying opinions exist amongst different researchers regarding the use of students and/or professionals as the

participants within studies.

Table 2-10 displays a sample of empirical research papers that have used students, industry professionals or both within their studies. The studies' conclusions and results vary and from those it is difficult to come to specific conclusions regarding the effectiveness or ineffectiveness of using students or industry professionals within research studies.

Table 2-10 Listing of articles discussing results of using students within empirical research.

Author	Objectives	Participants	Conclusions
Porter et al. (1995)	Tested different inspection methods.	Students	Scenario method returned best results.
Porter & Votta (missing citation)	Replicated previous experiment.	Industry	Students and industry results similar.
Höst et al. (Höst et al. 2000)	Factors affecting lead time.	Student & industry	Same under certain conditions.
Carver et al. (2003)	Why use students in research.	Students	Useful to test hypotheses with.
Runeson (2003)	Using PSP to measure improvement.	Student & industry	Same but more studies needed.

Table 2-9 and Table 2-10 demonstrate that using both students and professionals as participants within software engineering empirical research has advantages and disadvantages. A major advantage when using students is the homogenous nature of the participants in that an experience baseline can readily be established, each having successfully completed the same courses within their degree program. The anomaly within this, however, is that even though they may have completed the same sections of their degree courses, some may also have industry-based work experience.

Using industry professionals within a study is advantageous as results can be generalised to the wider community of developers. The anomaly introduced by professionals is the extent and variation of their experience within the area being studied. It is quite possible that a professional developer may have 15 years experience, for example, and yet may have very little or no experience within the bounds of what is being studied.

The results from two recent studies highlight the challenges posed when comparing variations in software engineering practices with participants having different experience levels. Dunsmore et al. (2002; 2003) conducted an OO code inspection, comparing Checklist-Based Reading with Use-Case Reading and Abstraction-Driven Reading techniques. The participants were final year undergraduate students.

The second study conducted by Thelin et al. (Thelin et al. 2003) was an inspection on OO requirements and design documents. The participants in this study were Masters students, many of whom had industry experience. The authors stated that they were "comparable to fresh software engineers in industry" (Thelin et al. 2003, p. 691).

2.11 Summary of the Literature Review

This section summarises some key values of the methodologies and techniques that have been developed and discussed within this chapter. The focus here is to explore the areas that still remain and how the shortcomings in these can be addressed. This will form the basis for the overall research within this thesis.

2.11.1 Strength of the state of the art approaches

A summary of the strengths in the following 4 areas is outlined below:

- The contribution of the software inspection techniques.
- The contribution of the software comprehension models.
- The contribution of Bloom's Taxonomy and its different applications.
- The contribution of the Think aloud data collection.

There are two key contributions that software inspections have had within the software development process:

- Their successful application has resulted in a significant drop in the number of defects that find their way into deployed software systems. This has been estimated to produce up to an 85% reduction in the number of defects (Fagan 1986).
- Software inspection techniques have provided software professionals with structured methodologies to inspect software artefacts specifically for defect detection.

Program comprehension research has contributed to the software development processes by:

- Providing an understanding into the five different cognitive models used by software developers as they attempt to understand the system they are working on in order to execute different software engineering tasks upon it.
- Providing the understanding that developers build their system knowledge generally through two different methods, through understanding the

functional behaviour of the system and/or understanding the control flow of a system.

The use of Bloom's Taxonomy is contributing to the research into programmer comprehension by:

- Providing a framework in which developer's cognitive levels can be categorised while they carry out a software engineering task.
- Providing a framework by which software engineering education tasks can be created, specifically to better enable developers to operate at the higher cognitive levels.

The application of protocol analysis in software engineering research has contributed to the field by:

- Providing a means to capture the thought process of software developers as they carry out different software engineering tasks.
- Providing the data that has been analysed giving much of the needed information to understand what is now known by researchers in how developers go about understanding a software system (see Table 2-8).

2.11.2 Weaknesses of the existing techniques, methods and approaches

In this chapter, we have carried out a detailed review of the literature pertaining to four areas, namely: Software Inspections, Programmer Comprehension, Bloom's Taxonomy, and the Think-aloud data collection method.

In the literature pertaining to software inspections and the different software inspection techniques, as discussed in Sections 2.2 and 2.4, a key weakness is that there is little understanding of the extent to which inspectors' experience levels impact upon the effectiveness of the inspection technique they are using. Different inspection techniques may be better suited for developers with more or less experience levels. Another key weakness is that there is little research into the cognitive levels that the different inspection techniques enable the inspectors to operate at while carrying out their inspection task, or what cognitive process model(s) they support.

In the literature on programmer comprehension models, as discussed in Section 2.5, a key problem is that program comprehension is a laborious, labour intensive human task that a developer must perform in order to be able to successfully carry out different software engineering tasks, be it maintenance, bug fixing or evolution upon a system. There are no silver bullets to resolve this, and continued research needs to find ways around and/or through the human bottlenecks Kothari talked about (Kothari 2008).

The literature has identified two ways in which Bloom's Taxonomy has been used to measure developer's cognitive levels: verb analysis (Xu, & Rajlich 2005) for think-aloud data and the use of the Context-Aware Schema (Kelly, & Buckley 2006) to categorise the think-aloud data. A key weakness of the verb analysis is that several of the same verbs appeared in multiple categories of Bloom's Taxonomy, thereby causing difficulty in classification. A key weakness identified within the Context-Aware schema is that it does not cater for the Synthesis/Creation category, level six

in Bloom's Taxonomy. This is because it was created to measure developers' cognitive levels while performing maintenance tasks. A second weakness identified with the Context-Aware schema is that when utterances are categorised in context, only the two previous utterances are considered. Without considering the utterances that follow, utterance categorisation may be inaccurate.

2.11.3 Research work that remains to be done

While the different methods and technologies discussed within this chapter have provided advances within the software development field, for today's demand of higher quality and more sophisticated and complex software systems, there are still a number of key issues yet to be addressed. In order to meet the demand for more sophisticated software systems, the following issues need to be considered to assist those developers who will build and maintain these systems.

The reading of software artefacts, and software artefacts written by others, is an essential skill that software professionals need to possess, yet it has generally been assumed that they possess this skill (Basili et al. 1996; Basili 1997; Gabriel, & Goldman 2000; Linger et al. 1979). Software inspections, which involve code reading, should assist developers to gain greater insight into and understanding of the code they are reading (Laitenberger, & DeBaud 2000; Kelly, & Buckley 2006). Yet software inspections as an explicit technique to improve the software developer's understanding of the system have not been researched in-depth.

The identification of software inspection techniques that enable inspectors to operate at higher levels of thinking, taking into account their level of experience, is

something that has not yet been carried out. Also, within this area, there has been little exploration of a possible correlation between defect detection and successful code modification.

To date, there has been no linking of software inspection techniques to program comprehension through a mapping of the inspection techniques that facilitate the different cognitive process models. Without this mapping and an understanding of the inspection techniques that facilitate higher levels of thinking, project managers, for example, are not able to implement the most effective inspection strategy to accommodate the various levels of experience that exist within a software development team.

2.12 Conclusion

This chapter presented a review of the literature pertaining to software inspections and different software inspection techniques that currently exist. Program comprehension and the cognitive models that currently exist within the literature were described and examined with a brief look at reverse engineering and software visualisation. Bloom's Taxonomy was considered in terms of its application in the research areas of software engineering, computer science and information technology. The Context-Aware Analysis Schema, a methodology for using Bloom's Taxonomy in measuring developers' cognitive levels was examined. A review was conducted regarding participants' experience levels within software engineering/computer science/information technology empirical research. Finally, a summary was provided of the key values and weaknesses of all methods discussed in this chapter. An indication has been provided of work that remains to be carried out.

Specifically, we are concerned with measuring the impact that software inspection techniques have upon developers' cognitive levels, and the benefits that knowledge derived from this may have upon both the training of software developers and the overall software development life cycle.

With this review as a basis, the following chapter will define several key concepts and the research issues that will be addressed within this thesis.

Bibliography

1028-2008 IEEE Standards for Software Reviews and Audits 2008, IEEE Computer Society.

Adelson, B. & Soloway, E., 1985, The Role of Domain Experience in Software Design, *Transactions on Software Engineering*, SE-11(11), pp. 1351-60.

Anderson, J.R., Farrell, R. & Sauers, R., 1984, Learning to Program in LISP, *Cognitive Science*, 8, pp. 87 - 129.

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

Anderson, L.W., & Sosniak, L.A. eds., 1994, *Bloom's Taxonomy A Fourty-Year Retrospective,* The National Society for the Study of Education, Illinois, U.S.A.

Aurum, A., Petersson, H. & Wohlin, C., 2002, State-of-the-art: software inspections after 25 years, *Software Testing, Verification and Reliability*, 12(3), pp. 133-54.

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

Basili, V.R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S. & Zelkowitz, M.V., 1996, The empirical investigation of perspective-based reading, *Empirical Software Engineering*, 1(2), pp. 133-64.

Bennet, K.H., 1990, An introduction to software maintenance, *Information and Software Technology*, 12(4), pp. 257-64.

Bergantz, D. & Hassell, J., 1991, Information relationships in PROLOG programs: how do programmers comprehend functionality? *Int. J. Man-Mach. Stud.*, 35(3), pp. 313-28.

Bloom, B.S. ed., 1956, *Taxonomy of Educational Objectives Cognitive Domain,* David McKay Company, Inc..

Booch, G., 1986, Object-oriented development, *Transactions on Software Engineering*, 12(2), pp. 211-21.

Briand, L.C., 2003, Software Maintenance and Reengineering, 2003 Proceedings Seventh European Conference on, *Software documentation: how much is enough?* pp. 13-5.

Brooks, R., 1983, Towards a theory of the comprehension of computer programs, *International Journal of Man-Machine Studies*, 18(6), pp. 543-54.

Brykczynski, B., 1999, A survey of software inspection checklists, *SIGSOFT Software Engineering Notes*, 24(1), p. 82.

Buckley, J. & Exton, C., 2003, Program Comprehension, 2003. 11th IEEE International Workshop on, *Bloom's taxonomy: a framework for assessing programmers' knowledge of software systems.* pp. 165-74.

Canfora, G., Mancini, L. & Tortorella, M., 1996, Proc. Fourth Workshop on Program Comprehension, *A workbench for program comprehension during software maintenance.* pp. 30-9.

Carver, J., Jaccheri, L., Morasca, S. & Shull, F., 2003, Proceedings: Ninth International Software Metrics Symposium 2003, *Issues in using students in empirical studies in software engineering education.* IEEE, pp. 239-49.

Curtis, B., 1980, Proceedings of the IEEE, *Measurement and experimentation in software engineering.* IEEE, pp. 1144-57.

Curtis, B., 1986, Empirical Studies of Programmers, *By the Way, Did Anyone Study Any Real Programmers?* Ablex Publishing, pp. 256-62.

Deimel, L.E. & Naveda, J.F., 1990, Reading Computer Programs: Instructor’s Guide

to Exercises, Carnegie-Mellon University.

Detienne, F., 1991, Reasoning from a schema and from an analog in software code reuse, *Empirical Studies of Programmers: Fourth workshop*, ESP91, p. 5.

Diehl, S., 2007, *Software visualization: visualizing the structure, behaviour, and evolution of software,* Springer Verlag.

Dunsmore, A., Roper, M. & Wood, M., 2000, The Role of Comprehension in Software Inspection, *The Journal of Systems and Software*, 52(2--3), pp. 121-9.

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

Dunsmore, A., Roper, M. & Wood, M., 2002, Proceedings of the 24th International Conference on Software Engineering ICSE '02, *Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection.* pp. 135-44.

Dunsmore, A., Roper, M. & Wood, M., 2003, The development and evaluation of three diverse techniques for object-orientated code inspection, *IEEE Transactions on Software Engineering*, 29(8), pp. 677-86.

Eick, S.G., Loader, C.R., Long, M.D., Votta, L. & Wiel vander, S., 1992, Proceedings of the 14th international conference on Software engineering ICSE '92, *Estimating software fault content before coding.* ACM, pp. 59-65.

Ericsson, K.A. & Simon, H.A., 1993, *Protocol Analysis,* The MIT Press.

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

Fagan, M.E., 1986, Advances in Software inspections, *IEEE Transactions on Software Engineering*, 12(7), pp. 744-51.

Fisher, C., 1987, Advancing the study of programming with computer-aided protocol analysis, *Empirical studies of programmers: Second workshop*, pp. 198-216.

Fjelstad, R.K. & Hamlen, W.T., 1979, Proceedings GUIDE 48, *Application program maintenance study: Report to our respondents.* Philadelphia, USA.

Forward, A. & Lethbridge, T.C., 2002, Proceedings of the 2002 ACM symposium on Document engineering, *The relevance of software documentation, tools and technologies: a survey.* pp. 26-33.

Gabriel, R.P. & Goldman, R., 2000, Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications, *Mob software: The erotic life of code.* Mineapolis, U.S.A.

Gilb, T. & Graham, D., 1993, *Software Inspection,* Addison-Wesley, Wokingham.

Glass, R.L., 1994, The software-research crisis, *Software*, 11(6), pp. 42-7.

Hartzman, C.S. & Austin, C.F., 1993, Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research CASCON '93, *Maintenance productivity: observations based on an experience in a large system environment.* IBM Press, pp. 138-70.

Höst, M., Regnell, B. & Wohlim, C., 2000, Using Students as Subjects - A Comparative Study of Students and Professionals in Lead--Time Impact Assessment, *Empirical Software Engineering*, 5(3), pp. 201-14.

Humphrey, W.H., 1995, *A Discipline for Software Engineering,* Addison-Wesley, Massachusetts.

Humphrey, W.S., 2000, *Introduction to the team software process,* Addison-Wesley, Massachusetts.

Jones, C., 1996, Software defect-removal efficiency, *Computer*, 29(4), pp. 94-5.

Kelly, T. & Buckley, J., 2006, 14th IEEE International Conference on Program

Comprehension (ICPC2006), *A Context-Aware Analysis Scheme for Bloom's Taxonomy.* pp. 275-84.

Kelly, T. & Buckley, J., 2009, IEEE 17th International Conference on Program Comprehension ICPC '09, *An in-vivo study of the cognitive levels employed by programmers during software maintenance.* pp. 95-9.

Khairuddin, N.N. & Hashim, K., 2008, ACS'08: Proceedings of the 8th conference on applied computer science, *Application of Bloom's taxonomy in software engineering assessments.* World Scientific and Engineering Academy and Society (WSEAS), pp. 66-9.

Knight, J.C. & Myers, E.A., 1993, An improved inspection technique, *Communications of the ACM*, 36(11), pp. 51-61.

Koenemann, J. & Robertson, S.P., 1991, Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology, *Expert problem solving strategies for program comprehension.* ACM, pp. 125-30.

Koerner, E.F.K., 1999, *Linguistic Historiography: Projects & Prospects,* John Benjamins Publishing Co.

Kothari, S.C., 2008, The 16th IEEE International Conference on Program Comprehension ICPC '08, *Scalable Program Comprehension for Analyzing Complex Defects.* pp. 3-4.

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

Lejter, M., Meyers, S. & Reiss, S.P., 1992, Support for Maintaining Object-Oriented Programs, *IEEE Transactions on Software Engineering*, 18(12), pp. 1045-52.

Letovsky, S., 1986, Empirical studies of programmers: First workshop, *Cognitive processes in program comprehension.* pp. 58 - 79.

Linger, R.C., Witt, B.I. & Mills, H.D., 1979, *Structured Programming; Theory and Practice the Systems Programming Series,* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. & Tulula, P., 1993, Proceedings of the IEEE Second Workshop on Program Comprehension, *Facilitating the comprehension of C-programs: an experimental study.* IEEE Computer Society, pp. 55-63.

Lister, R. & Leaney, J., 2003, Proceedings of the fifth Australasian conference on Computing education-Volume 20, *First year programming: let all the flowers bloom.* pp. 221-30.

Lister, R., 2000, On blooming first year programming, and its blooming assessment, *Australasian Computing Education Conference*.

Littman, D.C., Pinto, J., Letovsky, S. & Soloway, E., 1986, Mental models and software maintenance, *Empirical Studies of Programmers*, pp. 80-98.

Littman, D.C., Pinto, J., Letovsky, S. & Soloway, E., 1987, Mental models and software maintenance, *Journal of Systems and Software*, 7(4), pp. 341-55.

De Lucia, A., Fasolino, A.R. & Munro, M., 1996, Proceedings of Fourth Workshop on Program Comprehension, 1996, *Understanding function behaviors through program slicing.* pp. 9-18.

Martin, J. & Tsai, W.T., 1990, N-Fold inspection: a requirements analysis technique, *Commun. ACM*, 33(2), pp. 225-32.

Mayrhauser, A. & Vans, A.M., 1993, Proceedings of the IEEE Second Workshop on Program Comprehension, *From program comprehension to tool requirements for an industrial environment.* IEEE Computer Society, pp. 78-86.

Mayrhauser, A.V. & Vans, A.M., 1995, Program Comprehension During Software

Maintenance and Evolution, *Computer*, 28(8), pp. 44-55.

Mayrhauser, A.V. & Vans, A.M., 1996, Identification of dynamic comprehension processes during large scale maintenance, *Transactions on Software Engineering*, 22(6), pp. 424-37.

McCarthy, P., Porter, A., Siy, H. & Votta, L.G., 1995, Proceedings of METRICS, *An experiment to assess cost-benefits of inspection meetings and their alternatives: a pilot study.* p.100.

McConnell, S., 2004, *Code complete 2,* second ed. Microsoft Press,, Redmond, Wash.

Miller, G., 1956, The magical number seven, plus or minus two: Some limits on our capacity for processing information, *Psychological Review*, 63(8), pp. 1-97.

O'Brien, M.P., Buckley, J. & Shaft, T.M., 2004, Expectation-based, inference-based, and bottom-up software comprehension, *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6), pp. 427-47.

Oliver, D., Dobele, T., Greber, M. & Roberts, T., 2004, Proceedings of the sixth conference on Australasian computing education-Volume 30, *This course has a Bloom Rating of 3.9.* p.231.

Olofsson M, Wennberg M. Statistical usage inspection, in *New Reference* (1996).

Pennington, N., 1987a, Comprehension strategies in programming, *Empirical studies of programmers: Second workshop*, pp. 100-13.

Pennington, N., 1987b, Stimulus structures and mental representations in expert comprehension of computer programs, *Cognitive Psychology*, 19(3), pp. 295 - 341.

Pennington, N., Lee, A.Y. & Rehder, B., 1995, Cognitive Activities and Levels of Abstraction in Procedural and Object-Oriented Design, *Human-Computer*

Interaction, 10(2/3), pp. p171 -.

Pfleeger Lawrence, S. & Atlee, J.M., 2010, *Software Engineering Theory and Practice,* Fourth (International) ed. Pearson, Upper Saddle River, N.J.

Polya, G., 1957, *How to solve it,* Doubleday,.

Porter, A. & Votta, L., 1998, Comparing Detection Methods For Software Requirements Inspections: A Replication Using Professional Subjects, *Empirical Softw. Engg.*, 3(4), pp. 355-79.

Porter, A. & Votta, L.G., 1994, ICSE '94: Proceedings of the 16th international conference on Software engineering, *An experiment to assess different defect detection methods for software requirements inspections.* Sorrento, Italy, pp. 103-12.

Porter, A., Siy, H. & Votta, L., 1996, Software Process, volume 42 of Advances in Computers, *A Review of Software Inspections.* Academic Press, pp. 2074-2.

Porter, A., Votta, L.G.,.J. & Basili, V.R., 1995, Comparing detection methods for software requirements inspections: a replicated experiment, *Transactions on Software Engineering*, 21(6), pp. 563-75.

Potts, C., 1993, Software-engineering research revisited, *Software*, 10(5), pp. 19-28.

Pressman, R.S., 2005, *Software Engineering A practitioner's approach,* Sixth ed. McGraw--Hill, Boston,USA.

Rajlich, V., 1994, Proceedings of Summer School on Engineering of Existing Software, *Program Reading and Comprehension.* Bari, Italy, pp. 161-78.

Regnell, B., Runeson, P. & Thelin, T., 2000, Are the perspectives really different?--Further experimentation on scenario-based reading of requirements, *Empirical Software Engineering*, 5(4), pp. 331-56.

Rist, R.S., 1986, Empirical studies of programmers: First workshop, *Definition,*

Demonstration, and Development. pp. 28-47.

Robson, D.J., Bennett, K.H., Cornelius, B.J. & Munro, M., 1991, Approaches to program comprehension, *Journal of Systems Software*, 14(2), pp. 79-84.

Runeson, P., 2003, Proceedings 7th International Conference on Empirical Assessment and Evaluation in Software Engineering EASE'03, *Using Students as Experiment Subjects An Analysis on Graduate and Freshmen Student Data.* BCS Publishing, pp. 95-102.

Shneiderman, B. & Mayer, R., 1979, Syntactic/semantic interactions in programmer behavior: A model and experimental results, *International Journal of Parallel Programming*, 8(3), pp. 219-38.

Shneiderman, B., 1977, Measuring computer program quality and comprehension, *International Journal of Man-Machine Studies*, 9, pp. 465-78.

Shull, F., Lanubile, F. & Basili, V.R., 2000, Investigating reading techniques for object-oriented framework learning, *Transactions on Software Engineering*, 26(11), pp. 1101-18.

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

Sjøberg, D.I.K., Anda, B., Arisholm, E., Dyba, T., Jorgensen, M., Karahasonovic, A., Koren, E. & Vokac, M., 2002, Proceedings of the International Symposium on Empirical Software Engineering ISESE '02, *Conducting realistic experiments in software engineering.* IEEE, pp. 17-26.

Sjøberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N. & Rekdal, A.C., 2005, A Survey of Controlled Experiments in Software Engineering, *Software Engineering, IEEE Transactions on*, 31(9), pp.

733-53.

Soloway, E. & Ehrlich, K., 1989, Empirical studies of programming knowledge, , pp. 235-67.

Soloway, E., Pinto, J., Letovsky, S., Littman, D. & Lampert, R., 1988, Designing documentation to compensate for delocalized plans, *Communications of the ACM*, 31(11), pp. 1259-67.

Sommerville, I., 2007, Inaugural IEEE Conference on Digital EcoSystems and Technologies, DEST '07, 2007, *Design for failure: Software Challenges of Digital Ecosystems.* IEEE Computer Society.

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

Stasko, J., 1998, *Software visualization: programming as a multimedia experience,* The MIT Press.

Storey, M.D., Fracchia, F.D. & Uller, H.A.M., 1997, Journal of Software Systems, *Cognitive design elements to support the construction of a mental model during software visualization.* pp. 17-28.

Swanson, E.B. & Beath, C.M., 1989, *Maintaining information systems in organizations*.

2004, SWEBOK: Guide to the software engineering Body of Knowledge, *IEEE Computer Society: Los Alamitos, California*.

Thelin, T., Runeson, P., Wohlin, C., Olsson, T. & Andersson, C., 2002, 1st International Symposium on Empirical Software Engineering, *How much information is needed for usage-based reading? A series of experiments*. pp. 127-138.

Thelin, T., Runeson, P., Wohlin, C., Olsson, T. & Andersson, C., 2004, Proceedings

of the Software Metrics, 10th International Symposium on METRICS '04, *A Replicated Experiment of Usage-Based and Checklist-Based Reading.* IEEE Computer Society, pp. 246-56.

Thelin, T., Runeson, P. & Regnell, B., 2001, Usage-based reading--an experiment to guide reviewers with use cases, *Information and Software Technology*, 43(15), pp. 925 - 938.

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

Thelin, T., Runeson, P., Wohlin, C. & Andersson, T.O.A.C., 2004, Evaluation of Usage-Based Reading Conclusions after Three Experiments, *Empirical Softw. Engg.*, 9(1-2), pp. 77-110.

Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M. & Robbins, P., 2008, ACE '08: Proceedings of the tenth conference on Australasian computing education, *Bloom's taxonomy for CS assessment.* Australian Computer Society, Inc., pp. 155-61.

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

Tyran, C.K. & George, J.F., 2002, Improving software inspections with group process support, *Commun. ACM*, 45(9), pp. 87-92.

Vessey, I., 1985, Expertise in debugging computer programs: A process analysis, *Int. J. Man-Mach. Stud.*, 23(5), pp. 459 - 494.

Votta Jr, L.G., 1993, Proceedings of the 1st ACM SIGSOFT symposium on

Foundations of software engineering, *Does every inspection need a meeting?* p.114.

Votta, L.G., 1994, Proceedings of the Ninth International Software Process Workshop, *By the way, has anyone studied any real programmers, yet?* IEEE, pp. 93-5.

Weinberg, G.M. & Freedman, D.P., 1984, Reviews, walkthroughs, and Inspections, *IEEE Transactions on Software Engineering*, 10(1), pp. 68-72.

Weinberg, G.M., 1971, *The psychology of computer programming,* Dorset House Publishing,.

Whalley, J.L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P.K. & Prasad, C., 2006, Proceedings of the 8th Austalian conference on Computing education-Volume 52, *An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies.* p.252.

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

Wiedenbeck, S., 1986, Beacons in computer program comprehension, *Int. J. Man-Mach. Stud.*, 25(6), pp. 697-709.

Wilde, N. & Huitt, R., 1992, Maintenance Support for Object-Oriented Programs, *Transactions on Software Engineering*, 18(12), pp. 1038-44.

Wilde, N., Matthews, P. & Huitt, R., 1993, Maintaining object-oriented software, *IEEE Softw.*, 10(1), pp. 75-80.

Winkler, D., Halling, M. & Biffl, S., 2004, EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04), *Investigating the Effect of Expert Ranking of Use Cases for Design Inspection.* IEEE Computer Society, pp. 362-71.

Xu, S. & Rajlich, V., 2004, Proc. Third IEEE International Conference on Cognitive

Informatics, *Cognitive process during program debugging.* pp. 176-82.

Xu, S. & Rajlich, V., 2005, Proc. International Symposium on Empirical Software Engineering, *Dialog-based protocol: an empirical research method for cognitive activities in software engineering.* pp. 397-406.

Xu, S., Rajlich, V. & Marcus, A., 2005, Proc. Fourth IEEE Conference on Cognitive Informatics (ICCI 2005), *An empirical study of programmer learning during incremental software development.* pp. 340-9.

Yoon, B.D. & Garcia, O.N., 1998, Proceedings of the 4th Symposium on Human Interaction with Complex Systems, *Cognitive Activities and Support in Debugging*.

3.0 Chapter Three

Problem Definition

> Engineers today, like Galileo three and a half centuries ago, are not superhuman. They make mistakes in their assumptions, in their calculations, in their conclusions. That they make mistakes is forgivable; that they catch them is imperative. Thus it is the essence of modern engineering not only to be helped to check one's own work, but also to have one's work checked and to be up to check the work of others (Petroski 1992).

3.1 Introduction

Chapter One highlighted the importance of software in modern society as well as the evolution of the software development life cycle over the past 40 years or more. It touched on the different software engineering processes that have developed over time in an attempt to improve the quality of the software. Chapter Two reviewed the literature on software inspections and inspection techniques, programmer comprehension and cognitive process models, Bloom's Taxonomy and its application to software engineering research and think-aloud data collection and its use in software engineering research. The chapter showed the work that has already been done in those areas and concluded by identifying areas that have not yet been

investigated or that need further investigation.

This chapter's purpose is to articulate research issues that are yet to be investigated and/or issues that need further investigation. These form the basis for the problems and research questions that this thesis will address.

This chapter first outlines the issues that have been identified for further investigation. The terms that will be used within this thesis will then be defined. Finally, this chapter identifies the choice of research methodology that will be applied to solve the identified research issues.

3.2 Key concepts

The literature contains large amounts of information and differing definitions regarding the concepts that will be used throughout this thesis. Therefore, prior to entering into a discussion regarding the issues addressed within this thesis, it is important that the definitions used within be clearly identified. In order to clearly communicate the concepts discussed within this thesis, this section clarifies the definitions of the concepts used within this research.

3.2.1 Inspection

The ANSI/IEEE Std. 729-1983 defines an inspection as "a formal evaluation technique in which software requirements, design or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems" (IEEE, 1983).

"Inspection" within this thesis is defined as: *a process where software code is examined for verification and validation by one individual person who is not the code's author*. This definition clarifies how this term will be used in this thesis against the ANSI/IEEE standard by stating that the inspection is conducted by one individual person rather than by a group of people.

This definition is used based on the research carried out indicating that inspection meetings yielded no better result than individual inspections (Votta 1993; Porter, & Votta 1995; Laitenberger, & DeBaud 1997; Porter, & Votta 1998).

3.2.2 Defect

According to the *New Oxford American Dictionary*, a defect is "a shortcoming, imperfection or lack." In software a defect is often referred to as a bug[3]. When used within this thesis, a defect is considered to be: *where an artefact differs from both the implicit and explicit requirement specification. The deviation from the requirement specification may cause the system to fail, produce unexpected and/or incorrect results, or behave in unexpected ways. The unexpected results may cause minor failures, or may produce catastrophic failures*.

3.2.3 Code modification

Definition: *Code Modification is a change to the software code resulting in the software delivering existing functionality in a new, corrected, more efficient, more*

[3] The use of the term bug, in software, is generally ascribed to Rear Admiral Grace Murray Hopper's discovery of an error in the Mark II when she discovered a moth stuck in a relay.

maintainable or changed manner from prior to change being made.

Code modifications occur for a number of reasons throughout a software system's life time. It is important that when a developer makes a code modification, the modification does not affect any other part of the system in a negative way, that is introduce a defect or degrade the system's performance.

3.2.4 New functionality

Definition: *New functionality is where new code is added to an existing code base or existing code is reused, such that when the addition and/or reuse is completed the software posses functionality that it did not have prior to the addition of the new code or reuse of the existing code*.

This definition distinguishes itself from a code modification in that, once the new functionality has been added, the system performs a new task or new tasks. It is possible that changes are made to existing code and once this has occurred, the code possesses new functionality.

3.2.5 Software development process or life cycle

Processes are defined in many disciplines to carry out a series of steps intended to ensure that the product or service is correct and is fit for its intended purpose. These steps can then be repeated in order to produce another copy of the same product or service of the same quality. The steps executed to carry out this constitute a process. In creating a software product Pfleeger and Atlee (Pfleeger Lawrence, & Atlee

2010) point out that the process, from beginning to end, is often called a life cycle. Hence, the terms "software development process" and "software development life cycle" are often used interchangeably.

Definition: *a software development process is the series of steps carried out, or the structure placed upon the process to govern and guide the development of a software product*.

In the case of the Waterfall model, described in Chapter One, this is considered a software process model. The software process model demonstrates the way in which the software development process is carried out. The entire process from requirements through to maintenance is considered the life cycle.

3.2.6 Cognitive process model

Definition: *a Cognitive Process Model is the mental process a software developer implements when they go about the task of attempting to understand the software system they are currently, or are about to commence, working upon*.

Within this thesis, when the term cognitive process model is used, it refers to one of the five models explained in Chapter Two: Bottom-up, Top-down, Systematic, As Needed or Integrated.

3.2.7 Reading or inspection technique

In the context of this thesis, the terms Reading Technique and Inspection Technique

will be used interchangeably as they are considered to have the same meaning.

Definition: *a reading technique is a procedure implemented by a software developer or software inspector as they go about reading through a software system's code base in order to detect defects or gain an understanding of what tasks are implemented and how are they implemented by the code they are reading*.

3.3 General problems in software development

Software development projects are known to often fail to deliver what was expected, be behind schedule and over budget (Royal Academy of Engineering & The British Computer Society 2004). "Brooks Law" states that a project already behind schedule will run even further behind when new staff are added to the team (Brooks 1995).

Parikh and Zvegintzov (Parikh, & Zvegintzov 1983) state that a program that took one to two years to develop will consume between five and six years of maintenance time. Pressman (2005) reports that software maintenance consumes more than 60% of total effort in a software development project. Although these two accounts are slightly dated, Pfleeger and Atlee (2010) report that many newer surveys yield similar results and state that developers now often use the 80-20 rule: 20% of the effort in software development is consumed in the actual development, while 80% of the effort is consumed in program maintenance.

These figures indicate that a program with a 10-year life span will spend approximately eight years being maintained by the developers. Within that 10-year life span developers assigned to that project will come and go. The program's

original developers may have moved on to different projects, different roles or different companies. This poses problems for the development company, when a developer leaves they take their inherent system knowledge with them. It is reported that between 50% and 90% of the entire maintenance time is consumed by developers attempting to understand the code (De Lucia et al. 1996; Canfora et al. 1996).

Hence, the departure of experienced staff can be a very expensive occurrence. There is the initial loss from the time and money that has been invested in training the developer. There is also the loss of inherent system knowledge and understanding. The actual economic loss experienced by the company in the long term may be far greater as a result of the employee's departure than had the company offered a higher salary to the employee (if that was the reason for the departure).

The challenge the company faces here though is that keeping an employee on in order to keep the intellectual property may prove to be counter-productive. An unhappy or disgruntled employee can often lead to an unproductive employee. Unhappy and disgruntled employees can demoralise a development team thereby leading to the reduced productivity of an entire team. Software development organisations need to find ways to bring new developers up to speed on a project that is already in the production or maintenance phase.

3.4 Program comprehension as a key challenge

Program comprehension is an all-encompassing, and sometimes overwhelming task. From the days of the first software programs until today, with fourth-generation

languages, the understanding of a program, written by another, is a challenge all developers face. As programs increase in size, the program complexity usually increases with it. Windows 3.1 contained an estimated 4-5 million lines of code, while Windows Vista is estimated to contain 50 million lines of code (Hiner 2008) and Apple Macintosh OS X 10.4 (Tiger) contains 86 million lines of code (Jobs 2006).

The task of understanding a program in order to maintain, modify or add new functionality to it, is becoming more difficult for software developers, as shown by the increase in the estimated lines of code in modern operating systems. A single developer is unable to maintain this type of information in his/her head. Despite the different documentation that exists for a program, the only artefact that is guaranteed to be up-to-date is the executing code. Therefore being able to read code, and read it effectively is an essential skill all software developers must acquire.

With the continued development and sophistication of systems, it is no longer possible for a developer to sit down, starting at main, and read through the code line by line. For example, at a rate of one line of code per 30 seconds, it would take more than 80 years for a single developer to read through each line of code found in Apple Macintosh OS X 10.4.

Developers need tools, frameworks, guidance, and systematic strategies in order for them to acquire an understanding of the system on which they are working and to which they will make changes and enhancements.

3.5 Software inspection usage

3.5.1 Traditional use of software inspections

Traditionally, software inspections have been applied to detect defects within software artefacts. Figure 3-1 demonstrates the traditional software inspection process, with its pre-conditions and post-conditions, inputs and outputs. There are three general inputs in the software inspection process: the software artefacts to be inspected, the inspection technique that will be used, and the inspector's level of experience. The main outputs from the traditional software inspection process are a list of the detected defects. A secondary output from this process is a list of false positives. False positives are incorrectly identified defects, that is, a section or sections of the inspected artefact identified as containing a defect when in actual fact there is no defect.

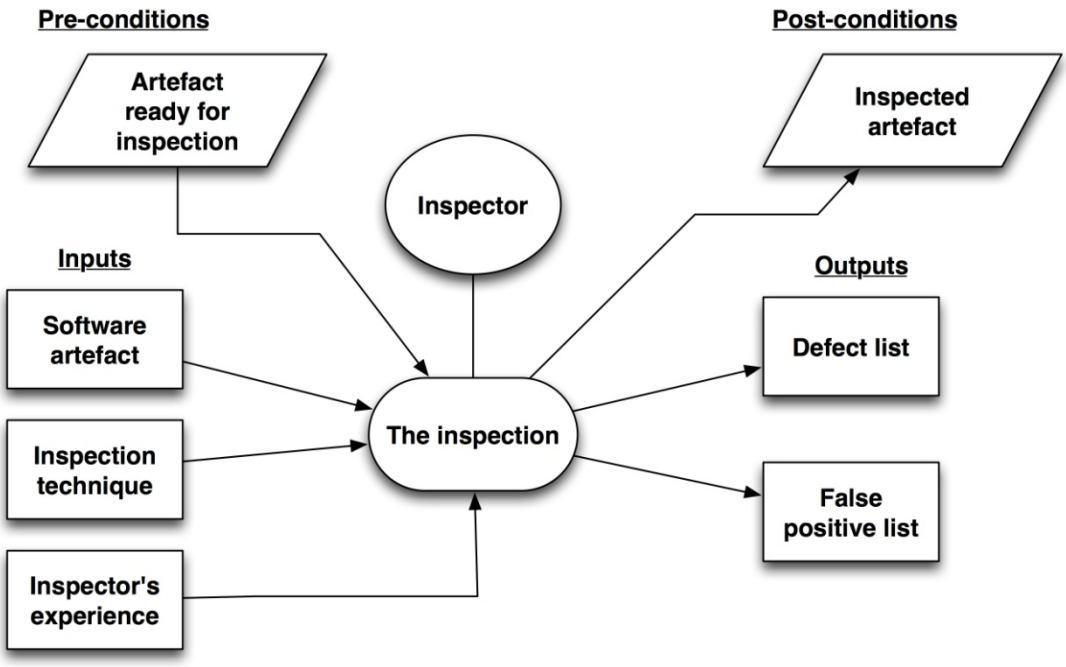


Figure 3-1. Traditional software inspection process.

The defect list is then used by a software developer to do the required work needed

to remove the defects. Once the re-work has been completed, the cycle is repeated with the corrected artefacts replaced in the system to be inspected again. Ideally, this cycle continues until all defects have been removed. However, the quality manager usually determines an acceptable level of estimated defects that remain in the code and authorises the product's release to the client.

3.5.2 Non-traditional software inspection application

In the context of this thesis, software inspections will be applied to a non-traditional application that specifically looks at the outcomes in the context of developer cognitive levels rather than that of defect detection.

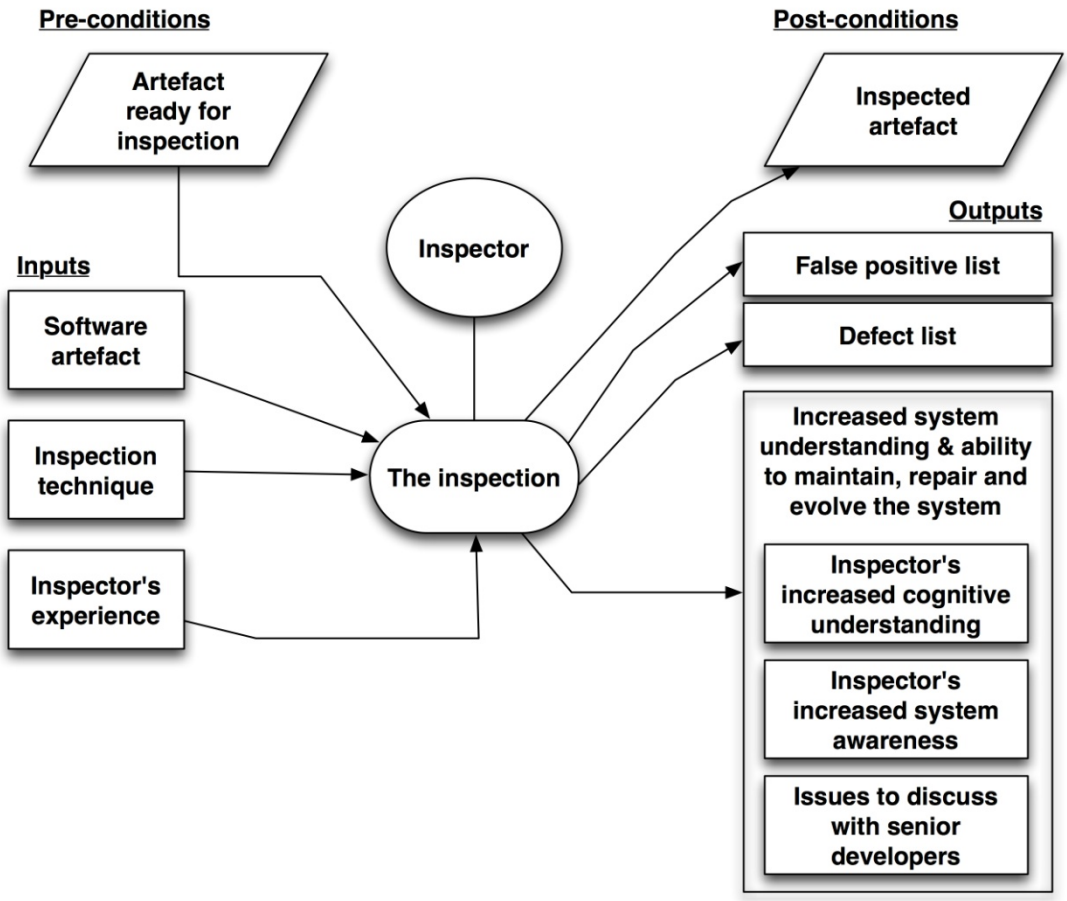


Figure 3-2. The inspection process for cognitive learning.

The software inspection's purpose within this context is to provide a software engineering process in which developers verbalise their thoughts and actions. These verbalisations are captured, categorised and analysed. The results from the capture, categorisation and analysis processes provide the information required in order to understand the cognitive levels at which participants operated while carrying out the inspection task. The software inspection technique becomes a variable within the process that will allow the identification of differences in these cognitive levels expressed by the developers that may be attributed to the varying software inspection techniques.

Figure 3-2 identifies outcomes from the inspection process that differ when compared with those of Figure 3-1. The two processes represented in Figure 3-1 and Figure 3-2 result in similar outcomes: the detection of defects within the artefact under inspection. One difference is, however, that the inspection represented in Figure 3-2 has outputs explicitly defined regarding the developer's cognitive level.

3.6 Problem overview

In this section, an overview of the problems related to the program comprehension and software inspections in the software development process is presented.

3.6.1 A need for a clear description of program comprehension

One problem that emerges from program comprehension research is its definition. Program comprehension and program understanding are often used interchangeably. The literature provides differing descriptions of program comprehension.

Providing a clear definition of program comprehension and where and how it applies will aid both software developers and software development teams. When developers articulate that they comprehend or that they understand the program, with these terms defined, the development team will be able to gauge where this developer may be able to fit into the team, and what tasks s/he will be able to perform.

The literature gives no clear and precise definition of software/program comprehension understanding. “Software comprehension is an incremental process to support the understanding of both the behaviour and the structure of software system” (De Carlini et al. 1993, p. 128). While this definition states that software comprehension is a process for understanding a software system, a clearer definition of program comprehension is required.

3.6.2 Problems on how to improve program comprehension

A second problem faced within the area of software development and programmer comprehension is how to increase developers’ understanding and comprehension of the software system on which they are working.

The discovery and/or identification of specific technologies that increase developer comprehension will assist both the developers and their project teams to create and maintain higher quality software systems. Reverse engineering and code visualisation are two technologies that have been used to assist developers to improve their level of understanding (De Carlini et al. 1993; Canfora et al. 1996;

Diehl 2007).

The literature review examined software inspections, noting that in order to carry out an effective inspection, the inspector needs to understand the code. However, the software inspection literature focuses on methods and techniques to detect defects within requirements, design documents, code and documentation. Generally, software inspection research focuses on improving existing inspection techniques, or creating and developing new techniques for inspecting software artefacts to detect defects.

Improvements within new or existing techniques are identified by an increase in the number of defects detected within the artefact under inspection, and/or the identification of the same number of defects in a shorter time period (Basili et al. 1996; Aurum et al. 2002; Dunsmore et al. 2001; Dunsmore et al. 2002; Dunsmore et al. 2003; Thelin et al. 2003; Thelin et al. 2004; Winkler et al. 2004).

However, the impact that different software inspection techniques have upon developers' cognitive levels during a code inspection has not previously been measured. Measuring this will provide new knowledge that this thesis will use to develop new processes for use during software maintenance.

3.6.3 No mapping from software inspection techniques to cognitive process models

As shown earlier in this thesis, software inspection techniques have been an effective and efficient technology to detect defects within software artefacts. What have not

been researched are the cognitive process models that the different software inspection techniques best facilitate when being applied by the software inspector.

The problem that exists without this mapping is that there is no means by which one can know the impact that a software inspection will have on a software inspector's cognitive understanding of the software artefact under inspection. Providing a mapping will assist software development team leaders and developers, to know which inspection technique to implement given the inspector's experience level and the level of understanding required about the software artefact

3.6.4 No mapping from software inspection techniques to Bloom's Taxonomy

Bloom's Taxonomy of Educational Objectives - Cognitive Domain, has been used within software engineering research to assist in understanding the cognitive levels developers work at when performing different tasks. However, the Taxonomy has not been applied within the software inspection area, to understand the cognitive levels inspectors operate at using different software inspection techniques. Without a mapping from Bloom's Taxonomy to the different inspection techniques, it is unknown at what level of Bloom's Taxonomy the inspection technique facilitates a software inspector to operate. A mapping from different software inspection techniques to Bloom's Taxonomy will provide software development teams with a means of identifying the software inspection techniques that may be best suited for both the task that needs to be performed and the experience level of the inspector carrying out the task.

3.6.5 No guidelines regarding when to use which inspection technique

There are currently no guidelines regarding which inspection technique should be used at different stages within both the software development life cycle or taught within the software engineering education programs. Understanding the cognitive demands that the various inspection techniques place upon users will provide an understanding of the most appropriate time to introduce the different inspection techniques into software engineering curriculums and training programs.

Guidelines into when different inspection techniques should be implemented by inspectors in order to better understand the software system they are working on will assist in the area of understanding. The guidelines will assist any decision-making regarding the reading technology which is most appropriate for current needs with respect to the tasks that need to be executed upon the software system. Without these guidelines, software development teams and software developers will be unsure about which technology to implement; an inappropriate choice may lead to lower cognitive levels, reduced understanding, and the failure to correctly execute the required task.

The third problem that emerges from the theories about how programmer comprehension is achieved is the measurement of developers' comprehension levels. This has proven to be a difficult task. The application of Bloom's Taxonomy will make it possible to measure the developers' cognitive levels during the inspection task and identify which if any, of the differing inspection techniques better enables developers to acquire system knowledge and understanding and make changes to the

system given their level of prior experience.

3.7 The key research question

The key research question here is how to create a reading methodology that will provide both a way for novice developers studying to become software professionals to operate at the higher cognitive levels from early in their training, and software professionals a means to also operate at the higher cognitive levels from the initial stages of their work on a new and/or existing software project.

Program comprehension literature has focused on software comprehension for the purpose of maintaining a program that has already been created. Different cognitive processes have been examined and identified. The strategies that developers utilise when attempting to understand a program that they will be required to maintain have been identified. These strategies include the top-down, bottom-up, systematic, as needed and integrated.

The modification of a program through the addition of a new functionality requires a different cognitive process. The developer must be able to see what the program currently does, understand what is required of the new functionality that must be added, and integrate this new functionality correctly so that it achieves what it is designed for, and yet not compromise the functionality of the existing program.

The experience level of a software developer will have an impact on the way s/he executes different tasks. Chapter Two identified different perspectives and interpretations regarding the use of students and industry professionals within

software engineering empirical research. In attempting to gain an understanding of this impact, this research will compare students with industry professionals performing a software inspection. There will be a comparison of the quantitative results and the comments made by both groupings, identifying differences in results and also perspectives of the task performed.

Chapter Two identified the benefits that have resulted over the years from the systematic application of software inspections within the software development life cycle. A knowledge and understanding of the program being inspected is assumed when inspections are being applied to a software artefact. Basili (Basili 1997) specifically noted that those who performed software inspections had a better understanding of the software they inspected than did those who tested the same software. He also noted that the inspectors were able to better estimate how many defects remained within the software after completing the inspection than did those who tested it.

In this manner, this research will carry out a study to investigate whether or not a correlation exists between the number of defects an inspector detects when performing a software inspection and the number of successful modifications that the inspector makes to that code base when adding new functionality.

As identified in Chapter Two, different code inspection techniques require the inspector to perform different tasks in order to locate the defect, while having the inspector focus on the software artefact from different perspectives. The focus of the different inspection techniques requires the inspector to apply different strategies to

understand the code. A sample of software inspection techniques, including each one applied within the research reported within this thesis, with the information known about the technique from the literature review in Chapter Two, will be mapped into the Bloom's Taxonomy level at which it is expected that the inspector will operate while implementing an inspection. Each inspection technique will also be categorised according to the cognitive models, described in Chapter Two, according to the most appropriate way in which it is to be applied by the inspector.

Moreover, this research will examine the cognitive levels that developers demonstrate while carrying out software inspection tasks. Examining and categorising these behaviours will provide the actual cognitive levels at which the developers are operating as they carry out their varying inspection-related tasks. Identifying and comparing the different cognitive levels expressed by developers using different inspection techniques will provide a means of ascertaining whether different inspection techniques facilitate different cognitive levels of operation when applied by developers.

The combination of results from the aforementioned steps will then facilitate the creation of a reading methodology that will provide a way for developers to operate at the higher cognitive levels from the initial stages of their work on a new and/or existing software project. These results may also lead to recommendations of reading techniques that can be incorporated into undergraduate, graduate and other training curriculums to provide developers with the skills needed to read software artefacts effectively. For this research, effective software artefact reading will be defined as reading and operating at the higher levels of Bloom's Taxonomy.

3.8 Research issues to be addressed

In order to answer the above key research questions, we will need to address the major research aims which are as follows:

- To identify the impact of prior experience on the perspectives and interpretations of software inspections as demonstrated by both novices and professional developers.
- To identify the cognitive levels expressed by software developers as they inspect and map a sequence diagram to the underlying code.
- To ascertain whether there is a correlation between the effectiveness of a software inspection and an inspector's ability to successfully add new functionality to the inspected software artefact.
- To identify the cognitive levels expressed by software developers as they carry out a software inspection.
- To identify the cognitive levels expressed by software developers as they carry out a task to add new functionality to a software system they have inspected.
- To develop a mapping of software inspection techniques to the underlying cognitive process model that they facilitate and a mapping to the Bloom's Taxonomy category that inspectors are expected to operate at.

3.8.1 Identifying the impact of developer experience

A software developer's level of experience impacts upon the way s/he carries out the

software development tasks. Professional and novice developers will have different perspectives and understandings about the same process. It is important to identify the impact that experience has on the perspectives and interpretations of software inspections and software inspection techniques in order to know the techniques that best suit developers with different levels of experience.

3.8.2 Identify cognitive levels expressed while mapping a sequence diagram to underlying code

Sequence diagrams are used within the object oriented software development process to identify and demonstrate the message passing that occurs between collaborating objects. Mapping the sequence diagram to the underlying code enables the developer to understand the code that is being used and also to check if the correct code and methods are being called. Identifying the cognitive levels at which these developers operate when executing this kind of task provides a means of knowing if and when this type of reading task is best suited to increase the developers' understanding of the software system on which they are working. It provides a means of knowing whether this technology is appropriate as a reading strategy to increase a developer's level of understanding.

3.8.3 Correlation between defect detection and adding new functionality

Software inspections have been shown to be an effective means of identifying and removing defects from software systems. It has also been implied that software inspections increase the inspector's understanding of the software system. If a

correlation is found between the effectiveness of software inspectors and their ability to successfully add new functionality to the inspected software artefact, this will support this research in that software inspections may be a means to explicitly assist developers to understand the system on which they are working. A direct side benefit of this practice would also be the possible increase in defect removal from the software system.

3.8.4 Cognitive levels expressed during a software inspection

Identifying the cognitive levels at which inspectors operate while carrying out software inspections using different techniques will help to identify the different cognitive levels that the different inspection techniques facilitate. This can then be used to identify the inspection techniques that require the inspector to operate at higher cognitive levels to aid with system understanding.

3.8.5 Cognitive levels expressed while adding new functionality

Adding new functionality to a software system places a cognitive load upon the software developer. Identifying the cognitive levels that software inspectors exhibit as they carry out this type of task will provide insight into the possible effects the software inspection and software inspection technique may have had on this task. Understanding the effect that the inspection technique had upon the cognitive levels the inspector operated at while adding new functionality may aid to develop software inspection/reading methodologies that better support developers when carrying out code changes to the system.

3.8.6 Mapping software inspection techniques to cognitive process models and Bloom's Taxonomy levels

Currently, no mapping exists that identifies the underlying cognitive process models that are facilitated by the different software inspection techniques. Also, there is no mapping between software inspection techniques and the different levels within Bloom's Taxonomy. These two mappings will enable both novice and professional developers to operate at higher cognitive levels, and can be incorporated into training and curriculums to introduce software inspections that require participants to operate at higher cognitive levels in order to produce higher quality software artefacts.

3.9 Research approach to answering the questions

Leedy and Ormrod (Leedy, & Ormrod 2005) state, "Research is the systematic process of collecting, analysing, and interpreting information (data) in order to increase our understanding of the phenomenon about which we are interested or concerned." The research process cycle, with each stage labelled is shown in Figure 3-3. The cycle becomes wider and wider, reflecting the increasing size of the body of knowledge.

The purpose of this thesis is to increase our understanding specifically regarding the research issues listed earlier in this chapter. To be able to do this, a systematic and established research methodology approach must be followed.

Therefore, the next sections will discuss the different research methodologies

available to this research, describe the one chosen, and explain the rationale behind this choice.

3.9.1 Research approaches

Predominantly, research methodologies are divided into two different categories. They are the:

1. Science and engineering approach; and
2. Social science approach.

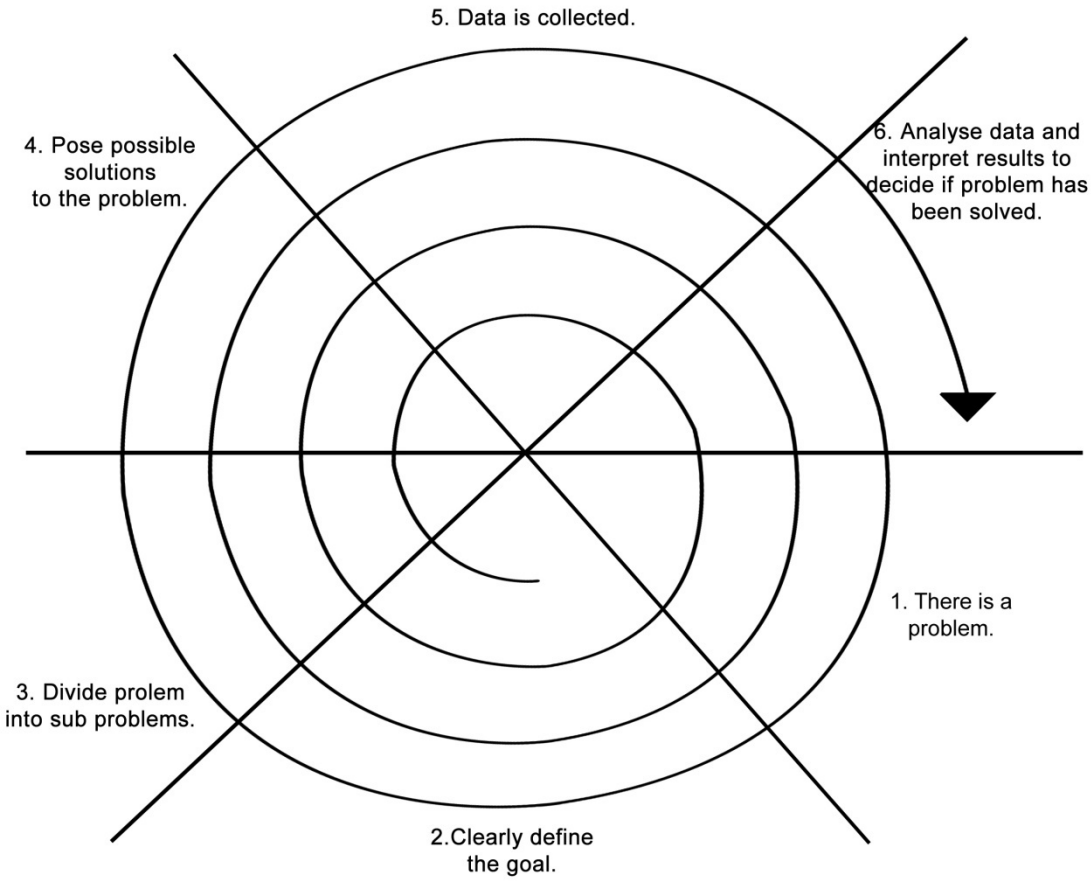


Figure 3-3. The research cycle.

The first approach, science and engineering, takes theoretical predictions and attempts to establish their validity. The goal of any endeavours in the engineering

field is to “make something work” (Galliers 1992). There are three levels considered when attempting to “make something work.” They are:

1. Conceptual level: creating new ideas and concepts (through analysis).
2. Perceptual level: formulating a new method and/or approach.
3. Practical level: executing testing and validation through experimentation.

The second general approach to research is the social science approach which can be broken down into two general sub-categories:

1. Quantitative; and
2. Qualitative.

Quantitative research is generally applied to research that involves the answering of questions regarding relationships that exist between measured variables. The goal is to predict, explain and control the phenomena. These results are then used for generalisations to other similar situations. They are used for theorising the existence, confirmation, or validation of relationships.

The term qualitative research is used across a wide range of different research techniques. The common factor in the different approaches is their focus on phenomena that occurs in the “real world” and the phenomena is studied in all of its intricacy (Leedy, & Ormrod 2005).

Qualitative research examines the decision-making in light of how and why. It is believed by many qualitative researchers that a single, all-encompassing truth may not be there to be discovered but rather, different individuals hold different positions

and each position may be equally valid or true (Creswell 1998; Guba, & Lincoln 1988).

"A main task (of qualitative research) is to explicate the ways people in particular settings come to understand, account for, take action, and otherwise manage their day to day situations" (Miles, & Huberman 1994). These investigations are generally carried out via the analysis of words. This is done by assembling and clustering the words that allows for comparison, contrasts, analysis and pattern identification (Juristo, & Moreno 2001).

Qualitative research usually deals with much smaller sample sizes, often involving structured or semi-structured interviews that can be analysed for all of their intricacies. The intricacies often arise during the interviews, allowing the researcher to pursue any new information that has emerged while data gathering. Kaplan and Maxwell (Kaplan, & Maxwell 2005) have argued that when qualitative data is quantified, the ability to understand the phenomena within its context is relinquished.

Quantitative research usually requires extensive data gathering and therefore a large sample size. The large sample size is needed in order to carry out different statistical analyses so as to be able to accept or reject given hypotheses. Finding a numerical relationship between different variables is the goal of quantitative research (Juristo, & Moreno 2001).

Whatever is to be examined within a study often will determine whether a qualitative

or want to take to the study will be carried out. Juristo and Moreno (Juristo, & Moreno 2001, p. 11) state that "it is the way in which the reality is described rather than the reality per se that is quantitative or qualitative."

Social science research, unlike engineering-based research, usually gives no explanation as to which methodology should be used or applied, nor how new problem-solving methodologies can be created. It is usually applied to test and evaluate methodologies which have already been produced by science and engineering.

3.10 Choice of research approaches

The previous section identified different ways in which research can be categorised. " Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use" (Sommerville 2007, p. 7) highlights two aspects of this definition:

1. "engineering discipline" in that engineers make things work.
2. "all aspects of software production" being not just the technical process but also the other activities involved in producing software.

In certain research situations, a multi-methodological research approach is necessary. This multi-methodological approach utilises the different research approaches as required to solve the research problems at hand (Nunamaker et al. 1991; Burstein, & Gregor 1999).

This thesis is grounded in the software engineering discipline and deals with making things work as well as other activities involved in producing software. This thesis will deal with the development of new methodologies for increasing developer comprehension levels and therefore requires the application of the science and engineering approach as well as the social science approach to research. Therefore, a combined research methodology will be applied within the scope of the research conducted for this thesis.

3.11 Conclusion

This chapter presented a summary of the problem statement that will be addressed by this thesis, and presented definitions of the key concepts, as they will be used within this thesis. The problem identified was systematically broken down into several different research issues to be addressed. Each issue was explained in relation to the existing literature. The chapter also defined the key concepts that will be used throughout this thesis. Following this, a short explanation of research approaches was given with the rationale for the choice of research approach that was adopted for our study.

The next chapter will present an overview of the conceptual framework being proposed as one way by which the identified problems may be addressed.

Bibliography

Aurum, A., Petersson, H. & Wohlin, C., 2002, State-of-the-art: software inspections after 25 years, *Software Testing, Verification and Reliability*, 12(3), pp. 133-54.

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

Basili, V.R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S. & Zelkowitz, M.V., 1996, The empirical investigation of perspective-based reading, *Empirical Software Engineering*, 1(2), pp. 133-64.

Brooks, F.P., 1995, *The Mythical Man-Month, anniversary edition,* Addison-Wesley, San-Francisco, USA.

Burstein, F. & Gregor, S., 1999, Proceedings of the 10th Australasian Conference on Information Systems, *The systems development or engineering approach to research in information systems: an action research perspective.* pp. 122-34.

Canfora, G., Mancini, L. & Tortorella, M., 1996, Proceedings Fourth Workshop on Program Comprehension, *A workbench for program comprehension during software maintenance.* pp. 30-39.

De Carlini, U., De Lucia, A., Di Lucca, G.A. & Tortora, G., 1993, Proceedings of IEEE Second Workshop on Program Comprehension, *An integrated and interactive reverse engineering environment for existing software comprehension.* pp. 128-37.

Creswell, J.W., 1998, *Qualitative inquiry and research design: choosing among five traditions,* Illustrated ed. Sage Publications, Thousand Oaks, California.

Diehl, S., 2007, *Software visualization: visualizing the structure, behaviour, and evolution of software,* Springer Verlag.

Dunsmore, A., Roper, M. & Wood, M., 2001, Proceedings of the 23rd International

Conference on Software Engineering ICSE '01, *Systematic object-oriented inspection - an empirical study.* pp. 135-44.

Dunsmore, A., Roper, M. & Wood, M., 2002, Proceedings of the 24th International Conference on Software Engineering ICSE '02, *Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection.* pp. 135-44.

Dunsmore, A., Roper, M. & Wood, M., 2003, The development and evaluation of three diverse techniques for object-orientated code inspection, *IEEE Transactions on Software Engineering*, 29(8), pp. 677-86.

Galliers, R., 1992, *Information systems research: issues, methods and practical guidelines,* illustrated ed. Blackwell Scientific Publications.

Guba, E. & Lincoln, Y.S. 1988, 'Do inquiry paradigms imply inquiry methodologies?', in DM Fetterman (ed), *Qualitative approaches to evaluation in education: the silent scientific revolution*, Praeger, New York , pp. 89-115.

Hiner, J., 2008, The top five reasons why Windows Vista failed, *ZDNET.* Retrieved December 15, 2009, from http://blogs.zdnet.com/BTL/?p=10303

Juristo, N. & Moreno, A.M., 2001, *Basics of software engineering experimentation,* Kluwer Academic Publishers.

Jobs, S., 2006, Keynote Presentation, World Wide Developers Conference WWDC '06, San Francisco, U.S.A.

Kaplan, B. & Maxwell, J.A. 2005, Health Informatics, in *Qualitative Research Methods for Evaluating Computer Information Systems,* Springer New York, pp. 30-55.

Laitenberger, O. & DeBaud, J.M., Perspective-based reading of code documents at Robert Bosch GmbH, *Information and Software Technology*, 39(11), pp. 781-

791.

Leedy, P.D. & Ormrod, J.E., 2005, *Practical research: Planning and design,* Eighth ed. Pearson/Merrill/Prentice Hall, Upper Saddle River, NJ.

De Lucia, A., Fasolino, A.R. & Munro, M., 1996, Proceedings of Fourth Workshop on Program Comprehension, 1996, *Understanding function behaviors through program slicing.* pp. 9-18.

Miles, M.B. & Huberman, A.M., 1994, *Qualitative Data Analysis,* Second ed. Sage Publications, London, UK.

Nunamaker, J.F., Chen, M. & Purdin, T.D.M., 1991, Systems development in information systems research, *Journal of Management Information Systems*, 7, pp. 89-106.

Parikh, G. & Zvegintzov, N., 1983, *Tutorial on Software Maintenance,* IEEE Computer Society.

Petroski, H., 1992, *To Engineer is Human,* Vintage Books, New York.

Pfleeger Lawrence, S. & Atlee, J.M., 2010, *Software Engineering Theory and Practice,* Fourth (International) ed. Pearson, Upper Saddle River, N.J.

Porter, A., Votta, L.G., & Basili, V.R., 1995, Comparing detection methods for software requirements inspections: a replicated experiment, *IEEE Transactions on Software Engineering*, 21(6), pp. 563-575.

Porter, A., Votta, L.G., 1998, Comparing detection methods for software requirements inspections: a replication using professional subjects, *Empirical Software Engineering*, 3(4), pp. 355-379.

Pressman, R.S., 2005, *Software Engineering A practitioner's approach,* Sixth ed. McGraw--Hill, Boston, USA.

Royal Academy of Engineering & British Computer Society, 2004, *The Challenges*

of Complex IT Projects, The Royal Academy of Engineering, London, U.K.

Sommerville, I., 2007, Inaugural IEEE Conference on Digital EcoSystems and Technologies, DEST '07, 2007, *Design for failure: Software Challenges of Digital Ecosystems.* IEEE Computer Society,.

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

Thelin, T., Runeson, P., Wohlin, C. & Andersson, T.O.A.C., 2004, Evaluation of Usage-Based Reading Conclusions after Three Experiments, *Empirical Softw. Engg.*, 9(1-2), pp. 77-110.

Winkler, D., Halling, M. & Biffl, S., 2004, Proceedings of the 30th EUROMICRO Conference EUROMICRO ’04, *Investigating the Effect of Expert Ranking of Use Cases for Design Inspection.* IEEE Computer Society, pp. 362-71.

4.0 Chapter Four

Solution Overview

4.1 Introduction

This chapter provides an overview of the way in which the research issues set out in Chapter Three will be addressed in this thesis. It provides a map for the remaining chapters of the thesis describing how each comprises part of the solution.

The previous chapters have described past scholarly work that has examined software inspections in the context of detecting defects to improve the quality of delivered software. These works have described the effectiveness of software inspections in the context of defect detection. The systematic application of software inspections has produced, over time, a reduction in the number of defects that make it into final release software. However, past work has not examined software inspections as a means of improving a software inspector's cognitive understanding of the system being examined.

In the previous chapter, we identified that improving software developers' understanding of the system on which they are working is an important research issue in software development and engineering. This issue was then divided into six smaller research issues.

In order to improve the software developers' understanding of the system, it is first important to clearly describe what is meant by "software developers understanding and comprehension of a system.

In this chapter, first, programmer understanding and comprehension will be clearly described. Having clearly described this, the remaining sections within this chapter will propose and describe the framework that will be applied within this thesis to solve the identified research issues.

4.2 Program comprehension

Today's software systems are complex and interconnected. Changes to a single module may impact upon the remaining system in an unpredictable manner. Prior to making changes to an existing system, whether it be corrections, modifications or adding new functionality, the software developer needs to adequately understand the system (Cornelissen et al. 2009).

Program comprehension and understanding has been described in the existing literature in the following ways: "Software comprehension is an incremental process to support the understanding of both the behaviour and the structure of the software system" (De Carlini et al. 1993, p. 128) and, "A person understands a program when he or she is able to explain the program, its structure, its behaviour, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program" (Biggerstaff et al. 1993, p. 27). These two descriptions are helpful in

presenting an overall picture of software comprehension and understanding. However, for the purposes of this thesis, the description will be further expanded using Bloom's Taxonomy. Prior to this, clear definitions of comprehension and understanding will be given.

The word "comprehension" is defined in the New Oxford American Dictionary (2005) as: *the action or capability of understanding something*. The word "understand" is defined as: *perceive the intended meaning, perceive the significance*. In carrying out a task such as a software inspection to verify that the system is operating correctly, the inspector must comprehend and understand the system. That is, s/he must know or perceive the intended meaning of the code and the task the code was intended to implement, and then verify whether or not the code carries out this intended task. The developer evaluates the code for correctness in relation to its intended purpose.

During a maintenance task, comprehension and understanding expand further because at this point, code may need to be corrected for defects, inefficiencies and omissions. There may also be new functionality requests from the client. The developer needs to understand and comprehend the system in order to successfully perform these tasks. An understanding of the program consists of being able to operate at higher cognitive levels of understanding depending on the required task.

By using Bloom's Taxonomy classification of cognitive levels as a reference, quantifiable parameters are placed around the classification. To successfully carry out the tasks described above such as correction, modification and adding new

functionality, the developer needs to operate at the three higher cognitive levels described by the taxonomy: Analysis, Evaluation and Synthesis.

- **Analysis**: where complex information is recognised and hidden meanings are revealed.
- **Evaluation**: where judgements regarding the work are made.
- **Synthesis** (also called creation): where something new is formed through the combining of different elements and parts.

When correcting a program, the inspector performs an Analysis of the code. The meaning behind things such as method calls and variables needs to be understood. The inspector then Evaluates the code to verify that the correct methods are called, and the correct operations are executed upon the variables. If the inspector has understood the code base, then this understanding facilitates seeing where and how the needed changes will impact on the overall system. Knowing this will allow the changes to be made with minimal system impact. Hence, to remove a defect or inefficiency, or to correct an omission, the inspector will need to operate at least at these two cognitive levels.

To add new functionality to the system, the inspector then operates at the Creation level. It is important that the developer be able to plan and design the required changes, evaluating the impact they will have on the wider system. Effectively, this requires operating at the Analysis and Evaluation levels. The inspector can then add the new functionality to the system, therby operating at the Synthesis level.

In the context of this thesis therefore, one goal is to identify and understand the software inspection technologies that require software inspectors to operate at these higher cognitive levels, as defined by Bloom's Taxonomy.

4.3 Conceptual solution framework

In this section, the conceptual framework is described, mapping out the way in which the identified research issues will be addressed within this thesis.

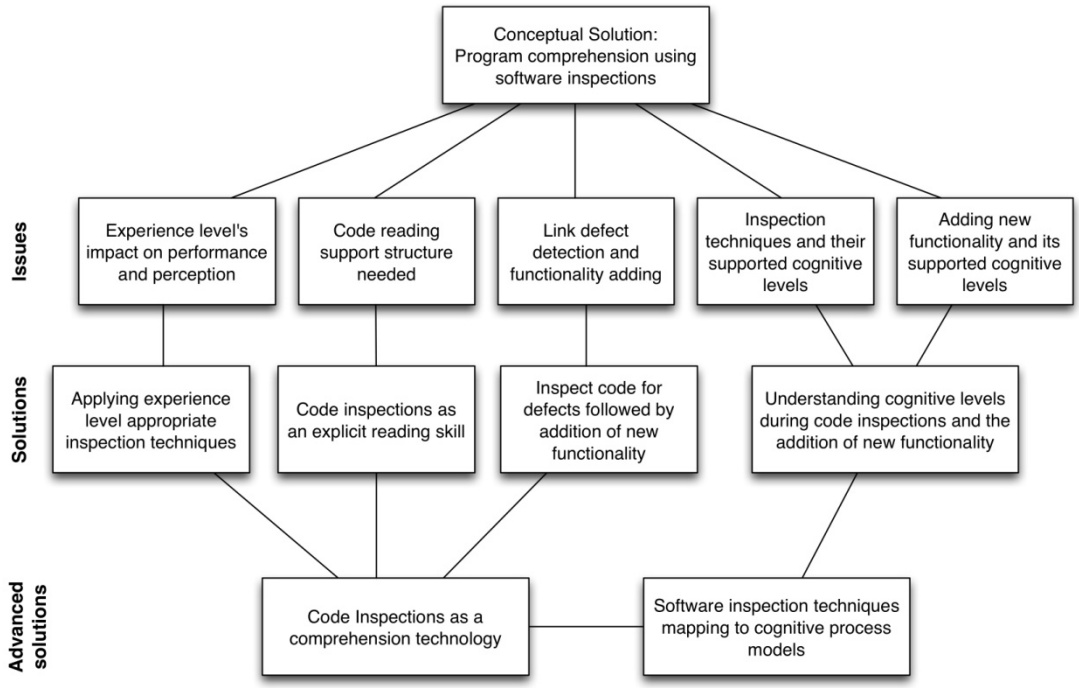


Figure 4-1. Conceptual solution framework of the research within this thesis.

Figure 4-1 depicts the conceptual framework with the identified research issues that will be addressed within this thesis and the solutions and advanced solutions that will be provided.

The box at the top of Figure 4-1 sets out the overall goal of this thesis: the

application of software inspections as a means to program comprehension. That is, to apply software inspections to improve developers' understanding of the code they are or will be working with.

The second row down in Figure 4-1 highlights the main issues that will be addressed by this thesis. Each issue will be addressed systematically in order to produce a solution within the context of this thesis.

The third row down in Figure 4-1 briefly identifies the solution being proposed to the identified issues. From these solutions, the advanced solutions will be derived. These advanced solutions provide a way by which the conceptual solution, intended to improve program comprehension using software inspections, is realised within this thesis.

4.3.1 Experience level's impact on performance and perspective

Intuitively, the more experience a software developer has, the better s/he will perform the tasks required out of him/her. Hence, experience can be seen as an obvious influence on the developer's effectiveness. However, this is not always considered a confounding factor within software engineering research, as is indicated by the differing reports shown in Chapter Two.

Our research will investigate the impact of level of experience on code inspection and maintenance activities. A comparison will be made between industry-based software developers and student developers, also referred to as novice developers. Comparisons will be made between the software inspection techniques used within

the investigations by the participants. The effectiveness of the inspection techniques will be examined using the number of defects detected by participants as well as feedback comments made by participants.

An analysis of this data will help to establish the degree of impact that experience has on a softer inspection, performed in a given situation.

4.3.2 Code reading support structure

When a software developer starts work on a new project, s/he needs a starting point. The ways in which developers build their understanding will also impact on their ability to carry out other tasks they will be required to perform on the system. Therefore, it is important that as developers go about understanding and comprehending a system, they have technologies that support them in this.

An experiment will be conducted in which developers will be given a code reading task. In this task, the developer will implement a UBR type of inspection. That is, they will carry out a UBR inspection requiring them to map the use case scenario depicted in the sequence diagram to the underlying code that executes the scenario within the program. Even though it is an inspection, in this instance the inspector will not be required to search for defects within the code.

For the duration of the task, the inspectors will “think-aloud” and this will be recorded for analysis. An online interface will also capture the way in which the inspector identifies the codes that execute the scenario.

This data will then be analysed to understand the cognitive levels demonstrated by the inspectors (participants) while executing this task. The correctness of the inspector's mapping will also be analysed. This information will provide an understanding of both the cognitive levels facilitated by this task as well as its effectiveness in assisting developers to read the code in order to accurately map the scenario to the underlying code.

4.3.3 Linking defect detection and the addition of new functionality

The successful completion of an inspection task implies that the inspector has achieved a certain level of understanding. Therefore, after the inspection, it is reasonable to expect that the inspector will be able to carry out successful changes to the code.

An experiment will be conducted in which student developers will be required to carry out a software inspection on a software system's code. The inspector's goal is to inspect the code and identify any defects that may exist within it. Following the completion of the inspection, the inspectors will be asked to add new functionality to the code base.

The data collected from this experiment will include: the number of defects detected and the number of successful modifications made by each participant. Also, the way in which the inspector goes about adding the new functionality will be captured by a screenshot taken every five seconds. The data will be analysed to identify any relationships that may exist between the number of defects detected and the number of successful changes made by the inspector.

A second group of new student developers will be presented with the software artefacts. They will not carry out an inspection, but will perform only the task of adding the new functionality to the code. A screenshot will also be captured every five seconds representing how they have gone about carrying out the task.

The ways in which the two groups go about performing the task will be compared. The differences will be analysed in order to understand the impact that previous exposure to the code has upon the way in which developers added new functionality to the code.

4.3.4 Cognitive levels supported by software inspection techniques

Each software inspection technique focuses the inspector on the code in different ways. For example, the CBR technique focuses the inspector by requiring yes or no answers to questions about the code. The UBR technique focuses the inspector by having him/her examine the code via prioritised use cases as they are executed through a sequence diagram.

These different focuses require an inspector to operate at different cognitive levels as s/he carries out the inspection. Understanding the cognitive levels required by the different inspection techniques will help to identify those inspection techniques that are the most appropriate for individual, hence different, inspectors.

To examine this, an experiment will be conducted in which novice software developers carry out a software inspection. The participants will be divided into

three groups. Each group will be assigned a specific inspection technique to use. Each participant will then carry out an individual inspection using the assigned inspection technique.

While carrying out the inspection, participants will "think-aloud." This will be recorded for analysis. A screenshot will also be taken every five seconds that will indicate what the inspector was looking at or doing to the code at that point in time. Participants will record the detected defects and answer a question sheet upon completion.

The data will be analysed to identify the cognitive levels at which participants were operating while carrying out the inspection. From the analysis, the cognitive levels required by the different techniques will be identified to understand which if any of the techniques better facilitated higher cognitive levels for novice inspectors, in a given situation.

4.3.5 Cognitive levels while adding new functionality

Adding new functionality to an existing software system requires the developer to understand the new functionality. The developer also needs to understand how this new functionality will be implemented, where it will be implemented, and how it might affect the wider system. As previously noted, this requires a developer to operate at different cognitive levels.

Having prior experience with the code may affect the developer. If a developer has previously carried out an inspection on the system and operated at various cognitive

levels during that task, this may impact on the developer's ability to add a new functionality to the system.

For this to be examined, an experiment will be carried out in which the student participants who carried out the software inspection previously, upon finishing the inspection, will be given a request to add new functionality to the system.

A new group of student participants, who have not previously seen or inspected the system, will also be given the new functionality request and be required to implement it within the system.

Participants will "think-aloud" as they implement the new functionality. A screenshot will also be taken every five seconds to see what the developer is looking at and/or working on. This data will then be analysed to understand the different cognitive levels the developer operated at. The data will be analysed, examining the different cognitive levels the developers operated at who had previously inspected the code using different inspection techniques. The cognitive levels of the developers who had not previously inspected or seen the system code will also be examined and compared with those who had implemented the inspection.

4.3.6 Inspection techniques and a cognitive level mapping

Software inspections have been well researched (Sjøberg et al. 2005) and shown to be an effective technology for detecting defects. Their application as a technology for improving developer cognition levels has not been researched. In using inspections to increase the cognitive levels at which developers operate, it is

important to identify two things: 1) the cognitive process model and the inspection technique that best supports it, and 2) using Bloom's Taxonomy classification, the cognitive level required by the inspection technique at which the inspector is required to function.

This knowledge can then be used by a software development team to make decisions about which inspection technique to implement for the goal they wish to achieve. Developers may also have preferred ways to go about learning a new system. Using this knowledge, developers will be able to select the inspection technology that suits them best.

An inspection technique mapping will be created using the acquired understanding from the literature review in conjunction with the data collected from the experiments. The inspection technologies tested within this thesis will be mapped to the cognitive process model they most closely support and to the Bloom's Taxonomy level classification at which an inspector is required to function.

4.3.7 Implementation of reading technologies as a comprehension strategy

Implementing a software inspection requires the inspector to read the software artefact. How to read code is often a skill that is "caught" rather than taught. Software Engineering, Computer Science, Information Technology and Information Systems' students are often expected to acquire this skill along the way.

A proposal will be made for the introduction and implementation of code reading technologies into degree and training programs to improve developers' ability to read and understand code. An improved understanding of the code will enable developers to carry out the task assigned to them whether it be to detect defects, maintain the code by making it more efficient, or evolve it by adding new functionality to it. The better the developers' understanding of the code, the more likely they are to carry out their task successfully, and without introducing new defects into the system.

4.4 Seeded defects

Within the experiments reported within, the seeded defects resulted from an examination of research literature identifying the types of defects being discovered within software. Also, the researcher used his personal experience of the defects often encountered when coding, as well as the defects that other researchers reported often encountering.

4.5 Threats to validity

In establishing the validity of research, two general questions are asked: First, are the results that have been drawn warranted, given the nature of the collected data as well as the way in which the data was collected? Second, from the results and conclusions drawn from the data analysis, can generalisations be made to the wider community, that is, the environment outside of the study? These two questions express the internal and external threats to which the research project is vulnerable to (Leedy, & Ormrod 2005).

4.5.1 Internal validity

A study's internal validity refers to the study's design, and the data collected from the research. The study design and the data collected must be appropriate in that they allow the researcher to draw conclusions regarding correlations and relationships within the data (Leedy, & Ormrod 2005; Trochim, & Donnelly 2006).

The internal validity threats to a study include: confounding, selection, history, maturation, repeated testing, instrument change, regression toward the mean, mortality or subject attrition, selection-maturation and experimenter bias. This is not an exhaustive list and it was noted that the titles given to these sometimes change.

The research reported within this thesis was subject to several internal validity threats. They were:

1. Selection;
2. Selection maturation;
3. Experiment bias; and
4. Instrument change.

Selection: is where the selection of participants is biased in order for the returned results to be more favourable or more significant in one particular direction.

With the studies reported within this thesis, several measures were put in place to limit this effect. When student participants were recruited, they were recruited in a general and open manner. An invitation on campus was issued to students enrolled in

the Bachelor degree programs of Computer Science, Software Engineering and Information Technology. The invitations were issued in several ways: posters were placed around the laboratories of the Department of Computing. The Computing Club displayed the posters within their meeting area, and a public announcement was made at a Computing Club meeting. The researcher also addressed the student body prior to several lectures.

Students were not approached on an individual basis, preventing the possibility that a selection bias could occur. Students who asked to participate and met the minimum requirements established, for the different studies, were accepted. No student who met the requirements was refused participation. In this manner, the threat of selection bias was reduced as much as was possible.

All studies reported within this thesis were conducted in the student's own time. It may be possible therefore that those students who did participate were the very eager and ambitious students, desiring to gain as much as possible from the university learning experience. They may not have reflected the entire student population. There is very little that a researcher can do to prevent this from happening and therefore this does need to be kept in mind when examining the results.

The industry professionals who participated in the studies reported in this thesis were approached in a different manner. There are several working relationships between industry and academia and different companies had indicated their willingness to participate in research studies. Therefore, the companies that had expressed an interest in participating in this kind of research were approached, and two companies

accepted the offer, making time available for their staff to take part. Also, several companies that could not participate as a group granted access to their staff by forwarding on invitation emails, or making verbal announcements regarding the research studies' need for professional developer participation.

By recruiting industry professionals in this manner, the selection bias threat for this participant group was also minimised.

Selection maturation: is the threat that occurs when there is a large difference in the maturity of participants. In this thesis, with the student participants, in order to limit this effect, minimum requirements were established and those requirements needed to be met in order for the student to take part. Demographics about student participants were also collected and students provided information about what, if any, software development industry experience they had had. This demographic information provided a way of identifying any anomalies in results arising from the levels of experience of the participants.

Demographics were also collected from the industry professional participants. The reason for this is similar to that of the students. Although one works in the software development industry, individual experience can be vastly different from that of others and even others on the same development team. This is further complicated, as some may have vast experience but nothing pertinent to what is required of them in this study. For example, it may be that the team manager has been working in the industry for 10 years, but for the last 8 years has been in a strictly managerial role, meaning s/he has not actually designed, developed or tested software during those 8

years. As with the student demographics, these were collected to monitor any anomalies appearing in the data so that they could be checked against the collected demographics.

Experimenter bias: occurs when the researcher behaves inconsistently with the participants. The researcher usually does this unconsciously, although it would be possible for the researcher to deliberately do this in order to alter the outcome of specific participants.

To reduce this validity threat, prior to the studies occurring, the researcher created sets of instructions. The instruction sets meant that each participant was given the same information as that given to the previous and the following participant. As the researcher needed to give verbal instructions with explanations, an instructions sheet was also created for the researcher. In this manner, every participant was told the same information, preventing both the omission of important information by the researcher simply forgetting to mention it, and also preventing the researcher from giving too much information to one participant and not enough to another.

Instrument change: is where the instrument used to measure the phenomena affects the data, resulting in a change in the process, thereby interfering with the data. It also can occur if the measuring instrument is changed during the study time, as this may be the cause of the results. When the measuring instrument is actually a human observer or analyser, then when they are more attentive, the results may differ from when they are distracted or not as attentive.

In some studies reported within this thesis, the researcher was required to categorise spoken language. In order to reduce the effect of the instrument change, that is the human instrument, a Kappa-Cohen statistic was generated. The Kappa-Cohen statistic is a statistical measure that is used to calculate inter-observer reliability. That is, two different independent researchers observe the same data and categorise it. This is done for several observations. The Kappa-Cohen statistic is then calculated to determine whether the two observers are categorising the data in a similar manner. Depending on the statistic that is generated, the reliability of the observers' categorisation is judged. With a high Kappa statistic, it then makes it possible for a single researcher to continue with the observations knowing that their categorisation is considered to be accurate.

In these studies, the data was categorised only after a satisfactory Kappa statistic had been generated. By doing this, the instrument change threat to validity was minimised.

4.5.2 External validity

A study's external validity can be divided into two parts: the population validity which relates to the findings that have been made and the possibility to generalise these findings to the wider community, and the ecological validity; that is, how results can be generalised given the environment established by the researcher in order to carry out the actual study (Leedy, & Ormrod 2005; Trochim, & Donnelly 2006).

These external threats to a study's validity include: experimentally accessible

population versus target population, explicit description of the experiment treatment, multiple treatment interference, Hawthorne effect, novelty and disruption effect, experimenter effect, pre-test sensitization, post-test sensitization, interaction of history treatment effect, measurement of the dependent variable and interaction of time of measurement and treatment (Bracht, & Glass 1968; Gall et al. 2006).

The studies reported within this thesis were subject to several external validity threats. These were:

1. Experimentally accessible population versus target population;
2. The Hawthorne effect; and
3. Measurement of the dependent variable.

Experimentally accessible population versus target population: was a threat because student participants were used. Student participants may not be reflective of industry professionals. Where students have been the participants, this has been clearly identified. Also, student participation was desired in order to measure novel developers, and developers who had not yet entered into full-time work within the software development industry. At no point throughout the work described in this thesis does the researcher attempt to generalise the data analysis results from student participants to industry professionals.

Industry professionals from the two companies that participated in the research described in this thesis were employed to build mission-critical software systems. There was a possibility that if the software they developed failed, people could be

severely injured or significant damage to property could occur.

Within the accessible population validity threat comes one of sample size. The think-aloud process produces an oral data set, in digital audio format for this research. The oral data needed to be manually transcribed. Current automated procedures do not provide accurate results. Once transcribed, it was categorised and analysed. These tasks are very time consuming, labour intensive undertakings. Therefore, as the participant number increases or the session length increases, the cost of the study increases significantly (Von Mayrhauser, & Vans 1995; Swarts et al. 1984). Sample sizes from studies reported in the literature have been: Studies reported within the literature have had sample sizes such as: 2 in Anderson et al. (1984), 3 in Adelson and Soloway (1985), 16 in Vessey (Vessey 1985), 6 in Letovsky (1986), 10 in Littman et al. (1987), 5 in Bergantz and Hassell (Bergantz, & Hassell 1991), 4 in Detienne (Detienne 1991), 2 in Kelly and Buckley (Kelly, & Buckley 2006) and 6 in Kelly and Buckley (Kelly, & Buckley 2009).

Given these constraints as well as the budget limits of this PhD research, the sample sizes have also been kept to resemble the sample sizes generally used for studies reported in the literature. However, Moore and McCabe (Moore, & McCabe 1999) point out that even with small sample sizes, significant differences can still be identified.

The Hawthorne effect: is the name given to the phenomenon that occurs when a person knows s/he is being observed, and his/her behaviour changes and generally improves. The name comes from studies conducted at the Hawthorne Works of the

Western Electric Company between 1924 and 1932. It was observed that when the workers knew they were being observed, their productivity increased. The Hawthorne effect is generally referred to when a study informs people that they are being observed in some way.

In the case of the studies conducted and reported within this thesis, the Hawthorne effect is noted in two areas: First, participants were required to think-aloud. This meant that they had a headset on for recording purposes and also they needed to verbalise their thoughts and actions. Second, participants were informed that screen shots of their current desktop was being taken every five seconds as they worked. The participants were aware that they were being observed and that the results of the tasks they were carrying out were going to be analysed.

Measurement of the dependent variable: relies on the distinguishing of the dependent variables. In the case of the artefacts being used within the studies, software defects were artificially inserted into the code and hence they may not reflect the defects currently encountered within the software industry.

4.6 Conclusion

Program comprehension is a key issue in the maintenance and evolution of existing software systems as well as in the development of new systems. In order to successfully carry out these tasks, the developer needs to be able to effectively read the software code.

In this chapter, program comprehension was described using Bloom's Taxonomy

classification of cognitive levels. The traditional use of software inspections was described, followed by a description of their non-traditional use as will be applied within this thesis.

The conceptual framework was presented along with a description of how each of the identified research issues from Chapter Three will be addressed. The threats to the validity of the experimental research results were also discussed.

The next chapter will examine the impact that experience has on an inspector's performance and perception of the software inspection and reading technique.

Bibliography

Adelson, B. & Soloway, E., 1985, The Role of Domain Experience in Software Design, *Transactions on Software Engineering*, SE-11(11), pp. 1351-60.

Anderson, J.R., Farrell, R. & Sauers, R., 1984, Learning to Program in LISP, *Cognitive Science*, 8, pp. 87 - 129.

Bergantz, D. & Hassell, J., 1991, Information relationships in PROLOG programs: how do programmers comprehend functionality? *Int. J. Man-Mach. Stud.*, 35(3), pp. 313-28.

Biggerstaff, T.J., Mitbander, B.G. & Webster, D., 1993, Proceedings of Working Conference on Reverse Engineering, *The concept assignment problem in program understanding.* pp. 27-43.

Bracht, G.H. & Glass, G.V., 1968, The External Validity of Experiments, *American Educational Research Journal*, 5(4), pp. 437-74.

De Carlini, U., De Luccia, A., Di Lucca, A. & Tortora, G., 1993, Proceedings of IEEE Second Workshop on Program Comprehension, *An integrated and interactive reverse engineering environment for existing software comprehension.* pp. 128-37.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L. & Koschke, R., 2009, A Systematic Survey of Program Comprehension through Dynamic Analysis, *Transactions on Software Engineering*, 35(5), pp. 684-702.

Detienne, F., 1991, Reasoning from a schema and from an analog in software code reuse, *Empirical Studies of Programmers: Fourth workshop*, ESP91, p. 5.

Gall, M.D., Gall, J.P. & Borg, W.R., 2006, *Educational Research: An Introduction,* Eighth ed. Allyn & Bacon.

Kelly, T. & Buckley, J., 2006, 14th IEEE International Conference on Program

Comprehension ICPC ‘06, *A Context-Aware Analysis Scheme for Bloom's Taxonomy.* pp. 275-84.

Kelly, T. & Buckley, J., 2009, IEEE 17th International Conference on Program Comprehension ICPC ‘09, *An in-vivo study of the cognitive levels employed by programmers during software maintenance.* pp. 95-9.

Leedy, P.D. & Ormrod, J.E., 2005, *Practical research: Planning and design,* Eighth ed. Pearson/Merrill/Prentice Hall, Upper Saddle River, NJ.

Letovsky, S., 1986, Empirical studies of programmers: First workshop, *Cognitive processes in program comprehension.* pp. 58 - 79.

Littman, D.C., Pinto, J., Letovsky, S. & Soloway, E., 1987, Mental models and software maintenance, *Journal of Systems and Software*, 7(4), pp. 341-55.

Moore, D.S. & McCabe, G.P., 1999, *Introduction to the practice of statistics,* third ed. W.H. Freeman and Company, New York.

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

Swarts, H., Flower, L.S. & Hayes, J.R. 1984, New Directions in Composition Research, in *Designing Protocol Studies of the Writing Process: An Introduction,* Guilford Publishers, pp. 53-71.

Trochim, W.M. & Donnelly, J.P., 2006, *Research methods knowledge base,* Third ed. Thomson Custon Publisher.

Vessey, I., 1985, Expertise in debugging computer programs: A process analysis, *Int. J. Man-Mach. Stud.*, 23(5), pp. 459 - 494.

Von Mayrhauser, A. & Vans, A.M., 1995, Program understanding: Models and

experiments, *Advances in Computers*, 40, pp. 1-36.

5.0 Chapter Five

The Impact of Experience on the Software Inspection Process

5.1 Introduction

The aim of this study is to explore the use of software inspections as a means of improving programmer comprehension – hitherto a non-traditional area of research. It proposes to do this by increasing the level of thinking at which a software inspector operates while carrying out the inspection. This chapter addresses the research issue regarding the impact that prior inspector experience has upon the software inspection process. This investigation compares the differences between the Checklist-Based Reading (CBR), Usage-Based Reading (UBR) and Use Case Reading (UCR) inspection techniques in identifying defects within the code and the impact that the level of inspector experience has on their efficacy.

5.1.1 The research issues

Software inspection techniques, as described earlier, are techniques used to inspect software code in order to identify defects. The changing software development environment has seen the rise of Object-Oriented (OO) projects over the past 15 years. This has resulted in a shift in development methodologies away from

procedural programming to the object-based programming paradigm.

The changing paradigm has led to questions about whether or not the software inspection techniques that are prevalent today, yet designed for the procedural paradigm, are still effective.

This experiment aimed to compare the CBR technique (which is considered the industry de facto inspection technique) (Laitenberger, & DeBaud 2000; Tyran, & George 2002), with the UBR and UCR techniques. The latter two techniques were specifically designed for inspecting OO systems.

The experiment compared these three inspection techniques to determine which inspection technique resulted in the highest number of defects being detected. It investigated the differences reported in prior research carried out by Dunsmore et al. (2003) and Thelin et al. (2003). The prior work by Dunsmore et al. compared three different software inspection techniques: CBR, UCR, and ADR. The participants were undergraduate students and they performed a code inspection. Thelin et al. compared the CBR and UBR inspection techniques on design documents with post graduate students, most of who had industry experience or had experience equivalent to that of an industry based junior software developer. The prior research was linked by controlling two independent variables found in the previous studies:

1. The reading technique implemented by each participant; and
2. The participant's level of experience.

5.1.2 Hypothesis

Based on results that had been reported within the literature regarding the effectiveness of UBR for defect detection, one hypothesis generated for this experiment was:

The null hypothesis, H_0, is described as:

> H_0: There is a significant difference between the UBR inspection technique and the CBR and UCR inspection techniques in detecting defects when applied by industry professionals.

The alternate hypothesis, H_1, is:

> H_1: There is no significant difference between the UBR inspection technique and the CBR and UCR inspection techniques in detecting defects when applied by industry professionals.

5.2 Methodology

This experiment is an extension of a previous experiment carried out by the author. In the earlier experiment, undergraduate student participants conducted a code inspection. The CBR, UBR and UCR inspection techniques were compared in order to identify whether there was a significant difference between the different inspection technique's defect detection yields. No significant difference was detected between the different inspection techniques for detecting defects. A small number of industry professionals also participated in that study. The results are reported in an Honours thesis (McMeekin, 2005).

In this experiment, the number of industry participants has been increased to

facilitate a defect detection yield comparison between the three different inspection techniques when used for a code inspection by students and industry professionals.

5.2.1 The study artefacts

In this experiment, the artefacts from the Dunsmore et al. (2002; 2003) studies were used. This enabled a comparison of inspection results returned from the code inspections using different inspection techniques. The code contained 14 seeded defects. Inspectors conducting checklist-based inspections used a composite checklist combining the salient features used by Dunsmore et al. (2002; 2003) and Thelin et al. (2003). The use cases were prioritised for the Usage-Based Reading technique.

The participants were each given the following artefacts:

1. A natural language specification;
2. A class specification;
3. A class diagram of the entire system;
4. The Java code to be inspected;
5. A defect reporting form and a feedback form;
6. Access to four other classes within the system;
7. Access to the Java API documentation;
8. Checklist (CBR inspectors only);
9. Sequence Diagram (UCR and UBR inspectors only);
10. Use cases (UCR and UBR inspectors only); and
11. Prioritised use cases (UBR inspectors only).

There were approximately 200 lines of code to be inspected, which falls within the accepted recommendations for a 120-minute inspection period (Fagan 1976; Gilb, & Graham 1993; Sommerville 2007). Participants were given an instruction sheet that described the inspection method they were to use. Participants performing the checklist inspection were given the checklist to use and participants carrying out the Use-Case and Usage-Based reading inspections were given use case scenarios, a sequence diagram and a scenario sheet. For the Usage-Based Reading inspection, the use cases were prioritised.

5.2.2 The participants

The participants in this study comprised both students and industry professionals. The students were enrolled in one of three Bachelor Degree programs: Computer Science, Information Technology or Software Engineering. The study did not form any part of their formal course work - it was voluntary and students participated in their own time. The base level requirements established for students to be eligible to participate in the study were: they were required to be in the third or fourth year of their degree, have successfully passed the two Java introductory courses and the first two Software Engineering courses.

These criteria for the student participants ensured a consistent experience baseline. It was established that all students had successfully completed these requirements and there was a common experience level established. This permitted the assumption that students had a similar base knowledge. There was still the possibility of differing experience levels within the student grouping and the impact that this could have had on the study.

Currently practising IT professionals from local industry also took part in this study. Table 5-1 shows the breakdown of industry participants with their corresponding years of experience using the different technologies applied within the study. The experience varied, as did the companies for which they worked now and in the past. For example, some participants had worked with companies that build mission-critical Java systems, enterprise Java systems and mobile phone applications using C++.

Table 5-1. Number of industry participants' experience levels in years with the different technologies.

	< 1 year	1 - 2 years	2 - 4 years	> 4 years
In industry	4	5	1	17
Working with Java	0	5	4	18
Working with UML	4	5	10	8
Working with OO	0	3	5	19

All participants within the two groupings had no prior knowledge of the system artefacts that they were inspecting.

5.2.3 Procedure

The task of inspecting the code was an individual one; hence participants were requested, during the inspection, not to interact with other participants. Given that students were participating during their own time, the student participants were also asked not to discuss the inspection with other students once they had completed the study. This was because the student to whom they were talking might participate at a later stage and hence this could change the outcome.

Participants were to note their start and finish times. Upon discovering a defect, participants also noted the time and wrote a brief description of the defect. The study did not require the participants to fix the defect, nor to describe how the defect might be fixed. Their task was only to identify the defects. The code under inspection was compiled and executed and participants were informed of this. Software inspections are a static task, and because of this, participants were required not to compile and execute the code themselves. In starting the inspection, the suggestion was made to all participants to commence by reading the natural language specification and after that to read the class descriptions of the system and its functionality. This was a suggestion only and participants were free to start in whatever manner they desired.

Participants conducting the CBR inspection were asked to answer each question on the checklist. Participants conducting the UCR and UBR inspections used the use cases and sequence diagram to guide them as they inspected the code. The UCR and UBR inspectors carried out the inspection in the manner described in Chapter Two. The UBR inspection participants were asked to consider the prioritisation of the use cases as they inspected the code.

5.2.4 Data collection and analysis

When participants identified a defect, it was entered into the defect-recording sheet. The class it appeared in, the line number or numbers where it appeared, and a brief description of the defect were entered into the description sheet. Prior to commencing the inspection, participants completed a brief demographic questionnaire. Upon completing the inspection, participants answered a second

questionnaire.

The collected data was analysed using the statistical packages R, Statistical Package for the Social Sciences (SPSS) and Microsoft Excel.

5.3 Results

A total of 26 industry professionals participated in the study: 8 completed the CBR inspection, 8 completed the UBR inspection and 10 completed the UCR inspection. There were 36 student participants in the study: 14 completed the CBR inspection, 12 completed the UBR inspection and 10 completed the UCR inspection.

Figure 5-1 displays the average number of times that each defect was detected by industry participants using the different inspection techniques. Figure 5-2 demonstrates the distribution of the results for the three different inspection techniques as applied by the industry participants.

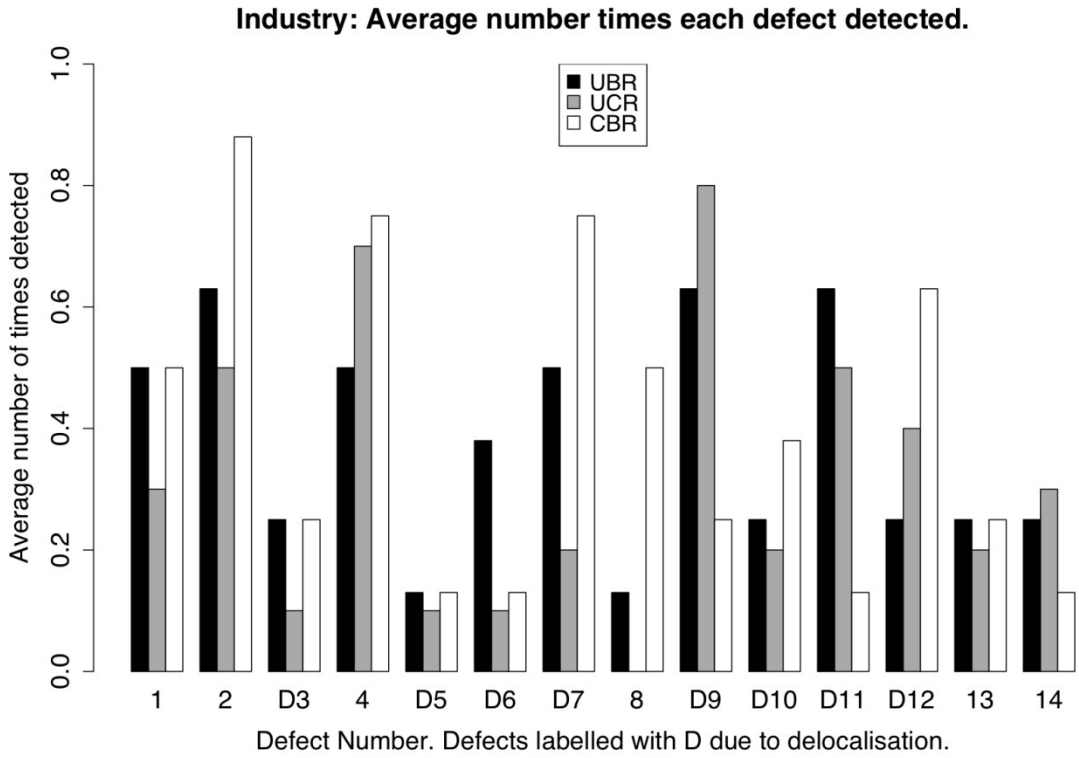


Figure 5-1. Industry participants: average number of times each defect was detected.

Figure 5-1 shows that no one specific inspection technique, as implemented by the participants, stands out as consistently detecting more defects than any of the other techniques. This is for both defects and defects due to delocalisation. The data displayed in Figure 5-2 indicates that there are no outliers within data set. The UBR technique data is the most broadly distributed of the three inspection techniques tested. The CBR and UCR techniques spread is less normally distributed than the UBR technique data.

The descriptive statistics for the overall defects detected, detected defects due to delocalisation and the false positives generated, are displayed in Table 5-2 for each inspection technique.

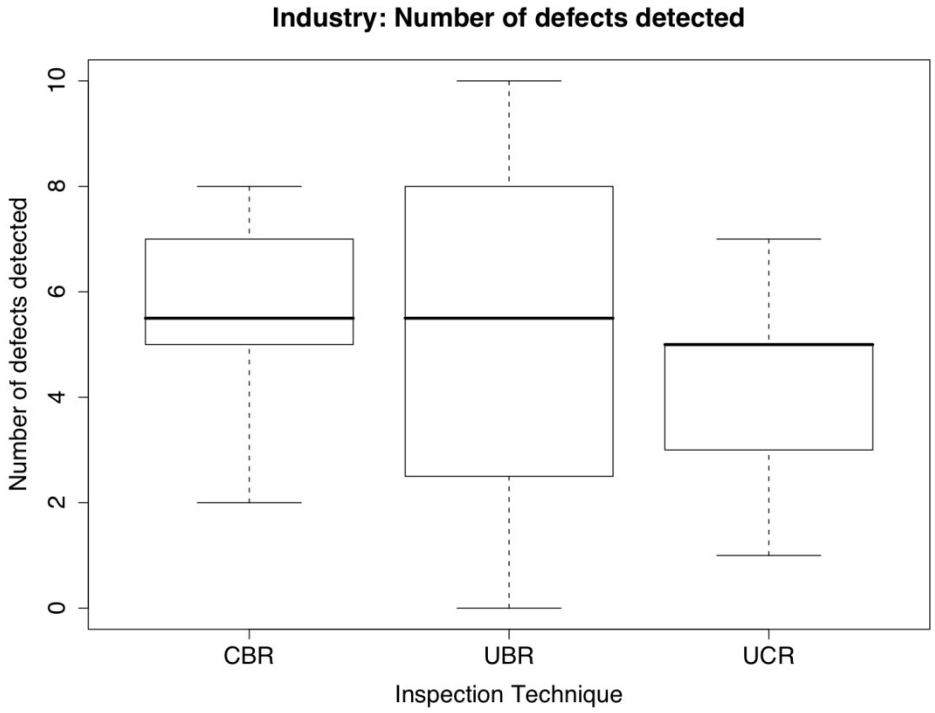


Figure 5-2. The spread of the number of defects discovered by each technique for industry participants.

Similar means were returned for the number of defects detected, the number of defects due to delocalisation detected and the number of false positives generated by the three different inspection techniques. The UCR technique returned zero false positives. However, both the CBR and UBR inspection techniques returned very low false positive means. In Figure 5-2, the three inspection techniques' median detection rates all lay close together.

Table 5-2. Descriptive statistics for industry participants.

		Inspection Technique		
		CBR	UBR	UCR
	Number of participants	***8***	***8***	***10***
Total defects (14)	**Mean**	5.6	5.2	4.4
	Std. Deviation	1.9	3.6	1.9
	Std. Error Mean	0.5	0.7	0.8
	Minimum	2	0	1
	Maximum	8	10	7
Delocalised defects (8)	**Mean**	2.6	2.9	2.4
	Std. Deviation	1.1	2.3	1.4
	Std. Error Mean	0.4	0.8	0.4
	Minimum	1	0	1
	Maximum	4	6	5
False positives	**Mean**	0.5	1.1	0.0
	Std. Deviation	1.9	1.9	1.7
	Std. Error Mean	0.5	2.8	0.0
	Minimum	0	0	0
	Maximum	2	8	0

In order to determine if there was a statistically significant difference between the three inspection techniques, a Kruskal-Wallis test was applied.

The Kruskal-Wallis test is a statistical test used to determine if significant differences occur when there is a single measurement variable and a single nominal variable that can have three or more values. The Kruskal-Wallis test does not assume the data to be normally distributed. It is the one-way anova test for non-parametric data (McDonald 2009).

In this case the single measurement variables in each test were: the number of defects detected, the number of defects due to delocalisation detected and the number of false positives generated. The nominal variable within these tests was the inspection technique used by the inspectors, CBR, UBR and UCR.

The first Kruskal-Wallis test was to determine if a statistical difference existed between the inspection techniques for the number of defects detected. The test returned an Asymptotic Significance of 0.4. This indicated that there was no significant difference between the three inspection techniques for the number of defects detected.

The second Kruskal-Wallis test was to determine if a statistical difference existed between the inspection techniques for the number of defects due to delocalisation detected. The test returned an Asymptotic Significance of 0.7. This indicated that there was no significant difference between the three inspection techniques for the number of defects detected due to delocalisation.

The third Kruskal-Wallis test was to determine if a statistical difference existed between the inspection techniques for the number of false positives generated. The test returned an Asymptotic Significance of 0.1. This indicated that there was no significant difference between the three inspection techniques for the number of false positives generated.

Therefore, the application of the three inspection techniques for defect detection neither outperformed nor under-performed each other in either hindering or aiding the detection of defects, the detection of defects due to delocalisation or in generating false positives when applied by the industry professionals.

Based on the evidence from the data collected from this experiment and the Kruskal-Wallis test to it, H_0 the null hypothesis is rejected and H_1 the alternate hypothesis is

accepted: there is a no significant difference between the UBR inspection technique and the CBR and UCR inspection techniques in detecting defects when applied by industry professionals.

5.3.1 Industry Professionals vs. Student Participants

The next step in this experiment was to compare the results from the industry professionals with the results from the student participants from the earlier research. The earlier research had returned similar results to these results; there was also no statistically significant difference found between the three inspection techniques when used by student participants in detecting defects, detecting defects due to delocalisation or in generating false positives.

The results comparing the three inspection techniques when used by industry professionals and student inspectors indicate that there were no significant differences. The hypothesis generated for this section of the experiment was:

The null hypothesis, H_0, is described as:

> H_0: There is no significant difference between the results of the industry professionals and the student inspectors.

The alternate hypothesis, H_1, is:

> H_1: There is a significant difference between the results of the industry professionals and the student inspectors.

Table 5-3 shows a breakdown comparing the data from the student participant grouping and the industry professional participant grouping. In regards to the three

metrics used, defects detected, defects due to delocalisation and false positives generated, the industry professional participants, using the three different inspection techniques, produced better results than did the student participants.

In order to determine if there was a statistically significant difference between the industry professionals and student inspectors, a Mann-Whitney test was applied.

The Mann-Whitney test is a statistical test used to determine if significant differences occur when there is a single measurement variable and a single nominal variable that can have only two values. The Mann-Whitney test does not assume the data to be normally distributed. It is the student's t-test for non-parametric data (McDonald 2009).

In this case, the single measurement variables in each test were: the number of defects detected, the number of defects due to delocalisation detected and the number of false positives generated. The nominal variable within these tests was the participant grouping the inspector belonged to: industry or student.

The first Mann-Whitney test was to determine if a statistical difference existed between the participant groupings for the number of defects detected. The test returned an Asymptotic Significance of 0.001. This indicated that there was a significant difference between the two participant groupings for the number of defects detected.

The second Mann-Whitney test was to determine if a statistical difference existed

between the two participant groupings for the number of defects due to delocalisation detected. The test returned an Asymptotic Significance of 0.02. This indicated that there was a significant difference between the two participant groupings for the number of defects detected due to delocalisation.

The third Mann-Whitney test was to determine if a statistical difference existed between the two participant groupings for the number of false positives generated. The test returned an Asymptotic Significance of 0.01. This indicated that there was a significant difference between the two participant groupings for the number of false positives generated.

Based on the evidence from the data collected from this experiment and comparing it with the data from the earlier experiment conducted using the Mann-Whitney test, H_0 the null hypothesis is rejected and H_1 the alternate hypothesis is accepted: There is a significant difference between the results of the industry professionals and the student inspectors.

Table 5-3. Mean of defects, delocal defects detected and false positives generated. Student vs. Industry.

	Inspection Technique		
	CBR	**UBR**	**UCR**
Number of participants	***14***	***12***	***10***
Students			
All defects	3.2	3.4	1.7
Delocal defects	1.5	2.2	1.0
False positives	2.0	0.8	0.8
Number of participants	***8***	***8***	***10***
Industry			
All defects	5.6	5.2	4.4
Delocal defects	2.6	2.9	2.4
False positives	0.5	1.1	0.0

5.3.2 Qualitative Data

Once participants had completed the inspection task, they provided feedback about the task they had undertaken. Among the industry professionals, more than 60% pointed out that both the code and diagrams they were supplied with contained no documentation or commenting of any sort. Under the definition of a defect established for this research and defined in Chapter Three, this was not a defect. The industry professionals stated that although it was not a defect according to the definition, for them it was nevertheless a very important defect and it would not have made it through their code production quality controls without documentation. In the student participant group, 17% made some reference to the lack of documentation but none believed it was a defect per se, but rather, an oversight. No student participant considered it to be a defect as the industry professionals had.

It was also noted by eight industry participants that IDE usage would have simplified the inspection, as it would have been easier to navigate through the documents used for the inspection. The industry professionals stated that IDE usage enabled them to be more productive with their time and facilitated navigation through documentation and code. Of the student participant grouping, none made any reference to IDE usage and its advantages. This more than likely arose from the fact that the students were not comfortable and also not as experienced with IDEs as they all tended to code using some kind of text editor.

Another noteworthy comment from the industry participants was their desire to be able to access testing data to aid and support the inspection. When questioned about this, none thought that the testing data would replace the inspection, but the testing

data would actually supply them with more information that would allow them to focus their effort on specific areas within the code that appeared to be causing problems. Again, no student participant made mention of or requested access to testing data. This may have also have been a result of students not having studied any unit/course on software testing, as there were none available within the degree programs being undertaken by the student participants.

The other comment made by industry participants was regarding the CBR inspection technique. They considered it to be very restrictive, focussing them on program structure rather than business logic. Those implementing it felt that it actually prevented them from inspecting the code as they would have liked to, as well as prevented them from continuing to "dig" into the code when they saw areas that may have contained defects or led to the cause of defects. Despite the fact that the quantitative data within the study showed that industry inspectors using CBR, on average, detected more defects than did other inspectors, they still considered it to be restrictive and not very helpful. The only exception to this was the one industry participant who used checklists in the work place, and did not feel this way. However, those who did not use them in their work place found them to be restrictive. The industry participants using the other two techniques did not find them restrictive at all.

The industry participants highlighted that both the UBR and UCR inspection techniques allowed defects to go undetected, as certain code was not included within the use case scenarios. They commented that another inspection technique would also need to be applied if all code were to be inspected. The student participants who

used these two techniques made no comment about code not being inspected by the application of the techniques. These comments by the industry professionals concur with Dunsmore et al. (2003) who state that the UCR technique does not cover all code, but needs to be implemented in conjunction with another technique to ensure that all code is inspected.

5.4 Conclusion

This chapter has shown that there is a significant difference between the results that student developers and professional developers return when they are the subjects in software engineering empirical research, in this case code inspections. This is an important issue for software engineering researchers to bear in mind as they continue to conduct research in this field. The research purpose needs to be clearly identified, and identified as to whether or not the results are going to be generalised to the wider community. If they are to be generalised, to what demographic of the community are they being generalised?

The study has also shown that the different code inspection techniques tested, did not produce significantly different results from each other. The CBR technique was not significantly better than the UCR or the UBR techniques, and neither of these two techniques was significantly better or worse than the other or the CBR technique. From the three techniques studied, no one technique was significantly better than any other technique in detecting defects, detecting defects due to delocalisation, or significantly worse by generating false positives. These study results were derived from experiments using both student participants and industry professional participants.

The qualitative data however, tells quite a different story. The student developers found that the UCR and UBR reading techniques to be quite difficult to follow. They struggled to fill out the scenario forms, as they were unsure of the possible outcomes from the things occurring within the sequence diagram and code implementation of it. The lack of guidance and structure was what, participants explained, made it difficult to work with those inspection techniques. The student participants who conducted the CBR inspection, on the other hand, found the technique to be very helpful. The participants stated that they felt they knew what to do and how to do it. The checklist gave them a great structure to follow and support them as they carried out the task at hand. The asking and subsequent answering of questions provided them with the structure they needed as well as a feeling of security in knowing what they needed to do and how they needed to do it.

The industry professionals were quite different from the students in the way they perceived the different inspection techniques. The UCR and UBR techniques were methods that none of them had used previously. As with the students, it was a novel experience. However, many reported that they found the CBR technique to be very restrictive. The need to respond to the series of questions gave them very little scope to "explore" the code. The tightly structured questions, they reported, made it difficult for them to check the code as they would normally do so .

Industry professionals indicated that with the checklist, they were unable to apply their personal experience to the code inspection. They were restricted to carrying out the checklist inspection and were not permitted to examine the code in a way that is

familiar to them and in a way that, when they have finished the task, they are confident that they have inspected the code to the best of their ability. The more structure provided to professional developers, the more they indicated they felt restricted in executing the code inspection. The structure hemmed them in, preventing them from carrying out the inspection in a manner with which they were familiar and which they believed to be successful based on their prior career experience.

Therefore, the results showed that the different inspection techniques did not assist developers, novel (student) or experienced developers (industry professionals) in the detection of more defects, more defects due to delocalisation, or the generation of more false positives. From these results, it can then be deducted that the inspection technique did not significantly assist participants to detect defects more effectively.

The results do indicate that novice/student developers prefer to have a structured methodology when working their way through code. The less structure provided, the more difficult the novice/student developers found it to carry out the software inspection. Their confidence level was low. The more structure they were provided with, the more "confident" they felt in the task that they had been given. The less structure they had, the more they felt insecure and lost as they attempted to complete the defect detection task.

For the industry professionals, however, the more structure they were provided with to carry out the inspection task, the more restricted they felt in executing the set task. They preferred less structure.

Hence, these results indicate that the inspection technique may not be a significant factor in the number of defects that inspectors detect while carrying out a code inspection. This supports the results shown in Dunsmore et al. (2003) regarding the influence that different inspection techniques had on the number of defects detected by an inspection. The qualitative data indicates that the inspection technique affects developers in other ways. For the industry professional, the highly structured inspection technique restricts them from being able to execute the task to the standard they want to. For the student, the highly structured inspection technique helps them to execute the task more confidently.

From these results, it appears that that inspection technique does not significantly increase effectiveness in defect detection. This leads to the question: is there a relationship between code inspections and other aspects of software development? Specifically in the context of this research, is there a relationship between the number of defects one detects and the number of modifications that a developer can make to the inspected code? Is there a relationship between inspection techniques and the cognitive levels at which a developer operates during the inspection?

The following chapter builds from here in examining whether a relationship exists between the numbers of defects one detects and the number of modifications one makes to that same inspected code base.

Bibliography

Dunsmore, A., 2002, 'Investigating effective inspection of object-oriented code', PhD. Thesis, University of Strathclyde.

Dunsmore, A., Roper, M. & Wood, M., 2003, The development and evaluation of three diverse techniques for object-orientated code inspection, *IEEE Transactions on Software Engineering*, 29(8), pp. 677-86.

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

Gilb, T. & Graham, D., 1993, *Software Inspection,* Addison--Wesley, Wokingham.

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

McDonald, J.H., 2009, *Handbook of Biological Statistics,* Second ed. Sparky House Publishing, Baltimore, Maryland.

McMeekin, D.A., 2005, 'A comparison of code inspection techniques', Hons. Thesis, Curtin University of Technology.

Sommerville, I., 2007, *Software Engineering,* Eighth ed. Addison-Wesley, Harlow.

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

Tyran, C.K. & George, J.F., 2002, Improving software inspections with group process support, *Communications of ACM*, 45(9), pp. 87-92.

6.0 Chapter Six

Inspectors' Cognitive Levels during a Usage-Based Reading Task

6.1 Introduction

In this chapter, the Usage-Based Reading (UBR) technique is implemented by inspectors so that the cognitive levels the technique facilitates can be examined. The software inspection technique, according to the results shown in the previous chapter, did not significantly impact upon the success or failure of the inspection task. However, industry participants reported that the CBR inspection technique was restrictive, not allowing them to use their own experience. However, the student inspectors found the CBR technique very helpful in guiding them through the inspection process.

The literature review in Chapter Two indicated that improvements within the software development process could come about through understanding, supporting and improving the way in which developers go about understanding a software system. Identifying efficient and effective means to do this is important in order for

the improvements to occur. A goal of this is the creation of tools, guidelines, documentation, and processes designed to facilitate the developers' cognitive processes in understanding and producing higher quality software systems.

In this chapter, the experiment examines developers' demonstrated cognitive levels as they carry out UBR task to map a sequence diagram to the underlying code. The participants had no previous knowledge or understanding of the system from which the scenario was derived.

This chapter introduces the use of *Bloom's Taxonomy of Educational Objectives, Cognitive Domain* (1956), in conjunction with the Context-Aware Analysis Schema (Kelly, & Buckley 2006) in order to measure developers' demonstrated cognitive levels as they carry out the UBR task.

6.1.1 Hypothesis

A sequence diagram is an interaction diagram within the Unified Modelling Language (UML). It displays communication, interaction and message passing between collaborating objects within a system. The diagram displays the execution of a scenario within the system (Booch et al. 2007; Page-Jones 2000).

Mapping a sequence diagram to the underlying code requires the developer to understand both what is displayed in, and what is represented by, the sequence diagram. The inspector must identify relationships and interactions between objects within different sections of the system, not simply within a single class. After understanding these representations within the diagram, the inspector must then map

these representations to the underlying code that will be executed when the scenario runs.

The mapping of the sequence diagram to the underlying code corresponds to the Analysis level of Bloom's Taxonomy. The Analysis level is described in the following way:

> Analysis emphasizes the breakdown of the material into its constituent parts and detection of the relationships of the parts and of the way they are organized (1956, p. 144).

Based on the reflection of how the task to map a sequence diagram to the underlying code so closely corresponds to the Analysis level within Bloom's Taxonomy, a hypothesis was generated for this experiment:

The null hypothesis, H_0, is described as:

> H_0: The inspectors will operate at the Analysis level more times than at any other cognitive level of Bloom's Taxonomy.

The alternate hypothesis, H_1, is:

> H_1: The inspectors will not operate at the Analysis level more times than at any other cognitive level of Bloom's Taxonomy.

Also, mapping a sequence diagram to the underlying code requires the inspector to start with an abstraction of a scenario within the system that demonstrates a specific piece of functionality. From this abstraction, the inspector maps this to the low level

code. In doing this, the inspector employs a top-down cognitive process model strategy.

6.2 Methodology

The participants' task for the purposes of this study was to map a use case scenario, displayed within a sequence diagram, to the underlying corresponding code. During the task, participants were required to verbalise their actions and their cognitive processes by using the "think-aloud" protocol, which was recorded.

The two means of data collection used for this study were:

1. The recording of the "think-aloud" data from each participant; and
2. An online interface, which was used to complete the sequence diagram mapping to the underlying code.

Data from the online interface provided access to the way in which participants mapped the scenario to the code. The data from the interface enabled observation of the steps participants took as they executed the task.

Transcribing and categorising participants' think-aloud data provided a means of identifying the cognitive levels participants demonstrated, according to Bloom's Taxonomy, while carrying out the required task.

6.2.1 The software artefacts

The software artefacts used for this study was code for a Java audio player system

with a command line interface. The software contained seven Java classes and consisted of a total of 330 lines of code. Participants were told that code compiled and executed without error. Figure 6-1 is the sequence diagram that participants mapped to the underlying code. Participants were also provided access to the following online artefacts:

1. The Use Case scenario;
2. The UML Sequence Diagram (Figure 6-1);
3. The interface in which to record which code was executed during the sequence diagram scenario;
4. The Java code; and
5. The Java API documentation.

6.2.2 Threats to validity

Chapter Four describes the threats to validity within the global context of this thesis. Specific to this experiment, an external threat to validity was the sample population size. A small sample size can prevent the results from being generalised to a wider population, as the sample may not be representative of the wider community. However, as has been noted, due to the nature of this research that used think-aloud data and the intensive labour cost to implement it, it has been generally accepted that these studies have much smaller sample sizes. For example, Anderson et al. (1984) had 2 participants, Adelson and Soloway (1985) had 3 participants, Letovsky (1986) had 6 participants, Bergantz and Hassell (1991) had 5 participants, Detienne (1991) had 4 participants and Kelly and Buckley (2006; 2009) had 2 and 6 respectively.

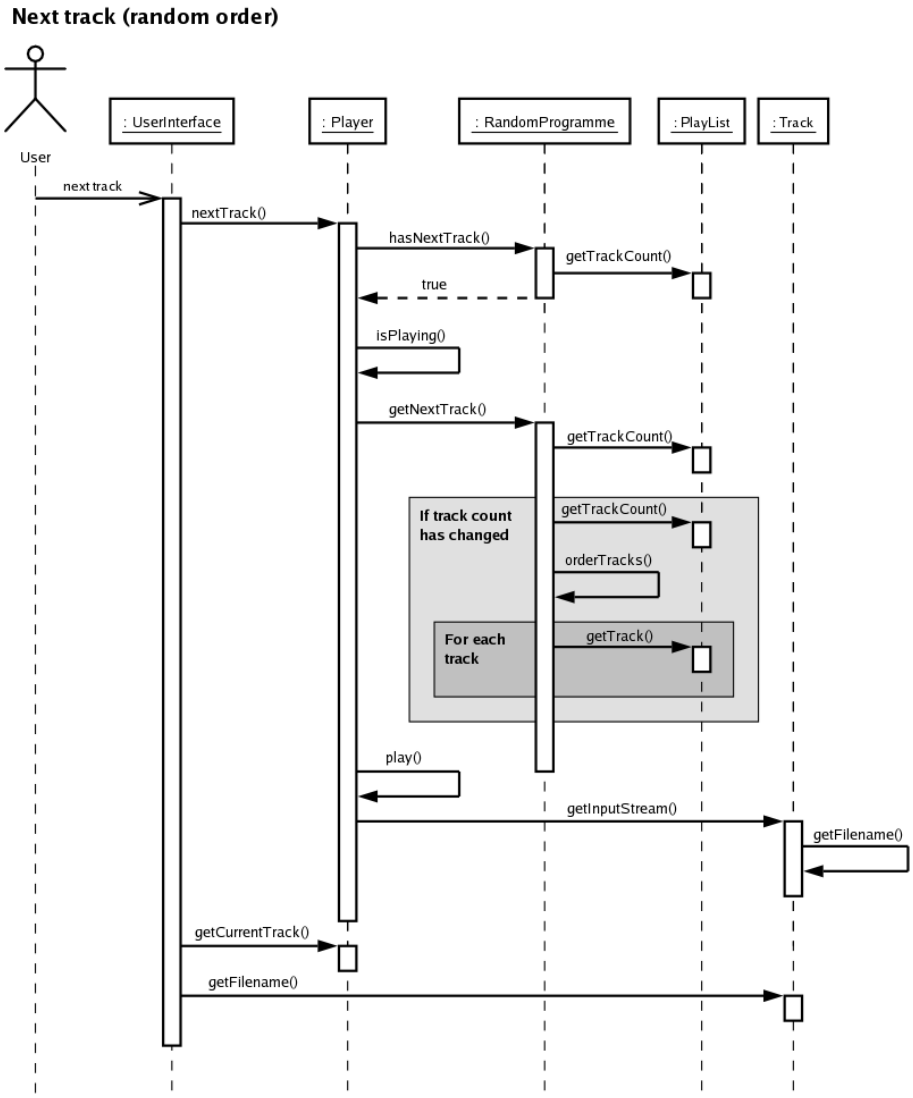


Figure 6-1. The Sequence Diagram to which participants were required to map code.

6.2.3 The participants

The study was advertised on campus throughout different courses within the three computing degree programs of Software Engineering, Computer Science and Information Technology, in order to recruit participants. Recruiting of industry participants took place through an email sent to companies which had participated in other empirical research and which had expressed interest in participating in further research. From the six participants recorded in the think-aloud data, two were from industry and four were final year undergraduate students enrolled in the Bachelor's

degrees in Software Engineering, Computer Science or Information Technology.

All participants took part during their own time. Student participants were informed that participation in the study would have no influence on the degree programs they were currently enrolled in and no part of the study was examinable within any course.

Industry participants were also informed that participation was voluntary and would have no impact on their employment.

6.2.4 Carrying out the experiment

Prior to carrying out the experiment, each participant took part in two small training exercises which were used to familiarise participants with the think-aloud protocol. Once they had successfully completed these exercises, they went on to the code mapping exercise. Prior to commencing the task, no participant had seen the software artefacts with which s/he was being presented. Participants had no prior knowledge of the system.

Figure 6-2 represents the use case scenario depicted within the UML sequence diagram participants were given. The user selects the “next track” option while the tracks are already playing in a random order.

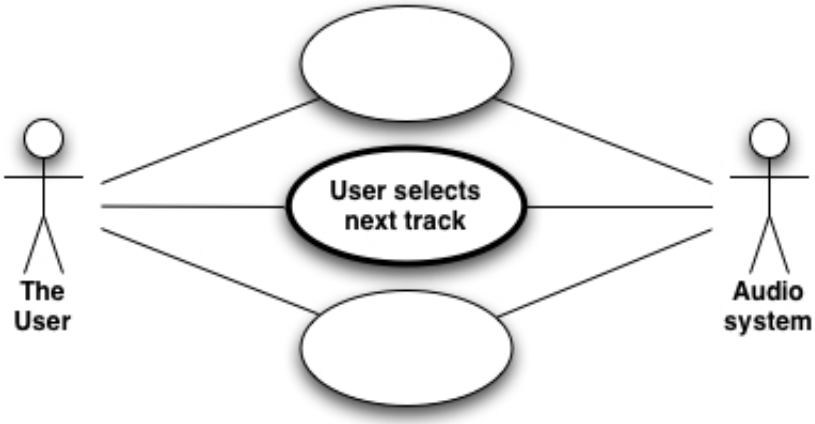


Figure 6-2. Use case diagram displaying the User selects next track scenario.

Ten participants took part in this study. A recording device failure caused the think-aloud data of four participants to be unintelligible; hence, it was not possible to transcribe and categorise that data into Bloom's Taxonomy levels. From the remaining six, two participants had industry experience and the remaining four participants were final year undergraduate students enrolled in either Computer Science or Software Engineering Bachelor degrees.

Each participant carried out the mapping task. The task required participants to map the scenario displayed in the sequence diagram (Figure 6-1) to the correct underlying code found in the seven Java files. Using the online interface, participants:

1. checked the box next to the lines of code in the Java files they thought were executed; and
2. listed the execution order of the lines of code that they thought were or may have been executed if the scenario was run.

If participants placed a question mark next to the line, this indicated that they thought the line might be executed. An asterisk next to a line indicated that the participant thought the line may be executed multiple times. Figure 6-3 demonstrates

this usage of the online interface. Several lines are listed in the interface; one line has an asterisk next to it and another line a question mark. The other four lines are ones that the inspector believes are definitely executed a single time.

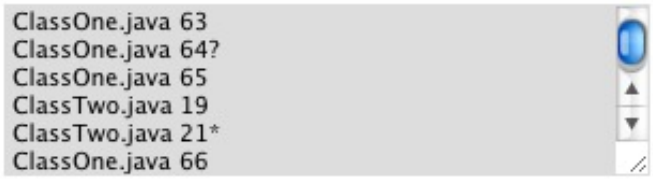


Figure 6-3. Example of the online interface for entering executed lines of code.

Figure 6-4 shows the checkbox system participants used to check if a line of code was executed. The example in Figure 6-4 shows the first five lines that would be executed if the scenario were run.

```
                try
                {
15 ☑                command = console.readLine( );

16 ☑                if( command == null )
                    {
17 ☐                    done = true;
                    }
                    else
                    {
18 ☑                    command = command.toLowerCase( );
19 ☑                    if( command.equals( PLAY ))
                        {
20 ☐                        player.play( );
21 ☐                        System.out.println( "Playing " +
                                player.getCurrentTrack( ).getFilename( ));
                        }
22 ☑                    else if( command.equals( PAUSE ))
                        {
```

Figure 6-4. Example of checking the boxes next to each line of code. The example shows the first 5 lines of code executed in the scenario.

No time limit was imposed and participants took as long as they needed to conduct the activity. When participants thought they were finished, they informed the researcher and the task was completed.

A model solution for mapping the sequence diagram to the corresponding code was independently created by the researcher and a second domain expert. Where discrepancies existed between the two researchers' solutions, consultations were held until the differences in interpretation were settled. Once this solution was created, a third domain expert was consulted in order to verify the model solution.

6.3 Results

After the think-aloud data from the participants was collected, it was transcribed and broken into sentences and/or utterances. Each sentence/utterance was mapped to a cognitive level in Bloom's Taxonomy. Utterances that were inaudible or totally unrelated to the study were listed in the Uncoded category. Table 6-1 lists an utterance example from one of the participants in each category of Bloom's Taxonomy, as well as an example from the Uncoded category.

Table 6-1. Utterance examples for each category from participants.

Bloom's Level	Example Utterance
Uncoded	*"can you actually hear me on the microphone?"*
Knowledge	*"program dot get next track program"*
Comprehension	*"and now you want ordered tracks which is coming from random program"*
Application	no utterance occurred at this level
Analysis	*"clip is a clip which is a private thing that I don't have access to"*
Evaluation	*"no it doesn't so my previous account of that was correct"*
Synthesis/Creation	no utterance occurred at this level

The author and one researcher, also familiar with categorising utterances according to Bloom's Taxonomy levels, independently coded the same 150 utterances using the Context-Aware Analysis Schema (Kelly, & Buckley 2006). The two sets of 150 utterances were compared and a Cohen's Kappa statistic was calculated to determine

the level of agreement between the two researchers' categorisations. The Kappa statistic was 0.63, which is considered a substantial agreement (Hartmann 1977). The author then coded the remaining utterances.

Table 6-2. Summary of the participants' utterances separated into Bloom's levels (Uncoded utterances are not included).

Bloom's Level	p1	p2	p3	p4	p5	p6	Mean
Knowledge	36%	41%	36%	23%	36%	41%	***36%***
Comprehension	44%	42%	41%	49%	43%	33%	***42%***
Application	0%	0%	0%	0%	0%	0%	***0%***
Analysis	18%	15%	18%	23%	19%	23%	***19%***
Evaluation	1%	2%	4%	4%	3%	3%	***3%***
Synthesis/Creation	0%	0%	0%	0%	0%	0%	***0%***

As the study's goal was to examine the utterances in the context of Bloom's Taxonomy, the Uncoded utterances have not been included in Table 6-2 and Figure 6-5. They are included in Figure 6-6 and Figure 6-7, as these show the timeline of participants' levels of thinking.

The categorisation of participants' utterances according to Bloom's Taxonomy are shown in Figure 6-5. The labels on the Y-axis, p1 – p6, represent each participant's ID, p1 = participant 1 etc. The graph shows that the Knowledge and Comprehension levels account for more than 70% of the participants' utterances. This accounts for a significant portion of the time taken by participants to perform the given task.

No participant expressed an utterance in the Synthesis level during the task; therefore, that level was omitted from the graph in Figure 6-5. The Synthesis level is where one creates something new or adds new functionality. In this experiment, participants were not required to carry out any task that fell into that category and no participant made an utterance appropriate for this category.

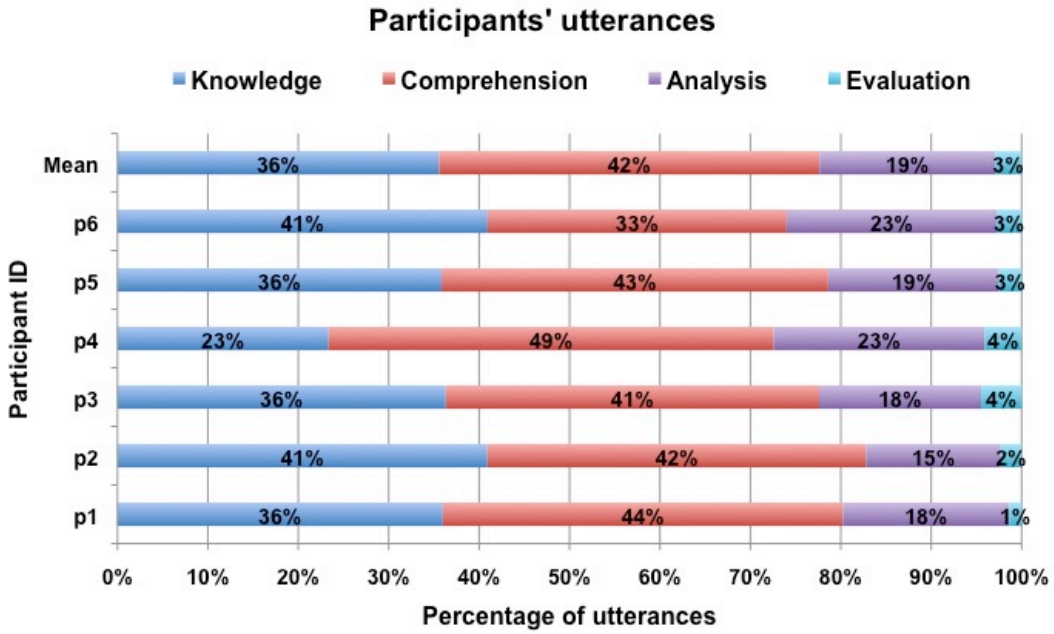


Figure 6-5. Participants' utterances broken down into Bloom's cognitive levels (Uncoded utterances are not included).

Participant six was the sole participant who expressed an utterance at the Application level. The Application level is where one is describing and making changes to the system. As was the case with the Synthesis level, participants were not required to perform any task that required them to function at this level. Even though participant 6 did make an utterance at that level, it accounted for less than 0.27% of the total utterances made by participant six; it did not appear in Table 6-2, and hence the Application level has been omitted from the graph in Figure 6-5..

Figure 6-6 and Figure 6-7 display the cognitive levels, in their expressed order, that participants six and four verbalised while carrying out the mapping task. The vertical axis reflects the Bloom's taxonomy levels and the horizontal axis shows the order in which they were expressed. The graphs of participant four and participant six were arbitrarily chosen as examples.

Table 6-2 lists the total percentages of each participant's utterances, broken into Bloom's categories. Figure 6-6 and Figure 6-7 depict the timeline of the cognitive levels the participants were demonstrating as they mapped the sequence diagram to the underlying code. These two graphs, together with the table, demonstrate that the participants operated, for the vast majority of the time, at the Knowledge and Comprehension levels. There were also small time periods when participants operated at the Analysis level and on a small number of occasions also at the Evaluation level.

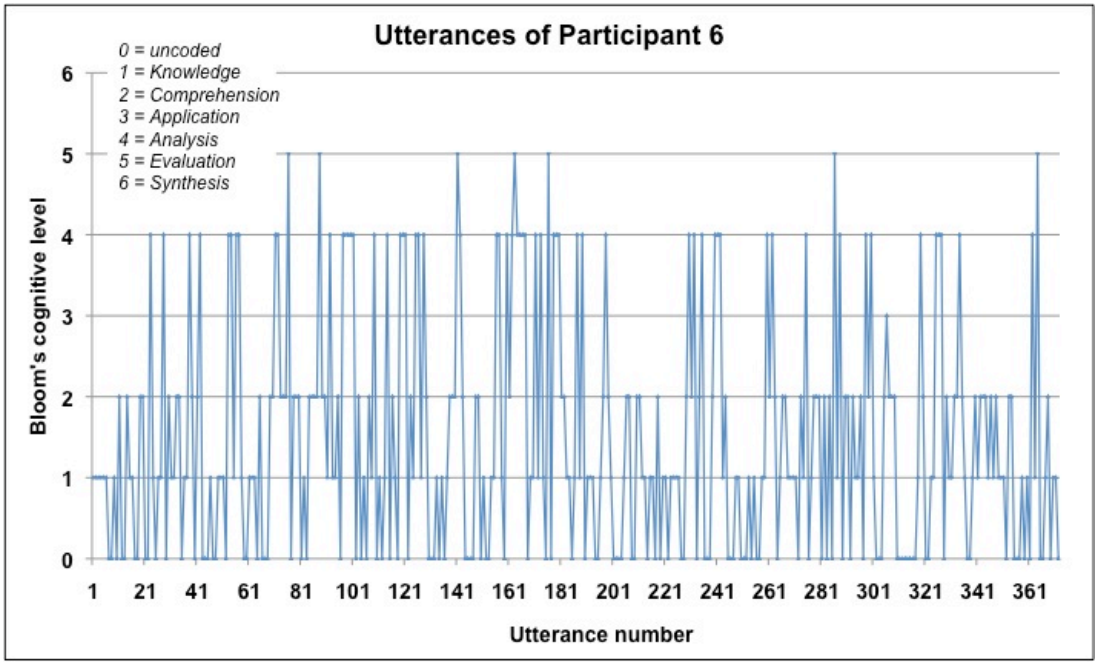


Figure 6-6. Participant 6's utterances, Bloom's cognitive levels in occurrence order.

Given the results of this experiment and with this sample size, the null hypothesis set out in Section 6.1.1 H_0: the inspectors will operate at the Analysis level more times than at any other cognitive level within Bloom's Taxonomy, is rejected and the alternate hypothesis H_1: the inspectors will not operate at the Analysis level more times than at any other cognitive level within Bloom's Taxonomy, is accepted.

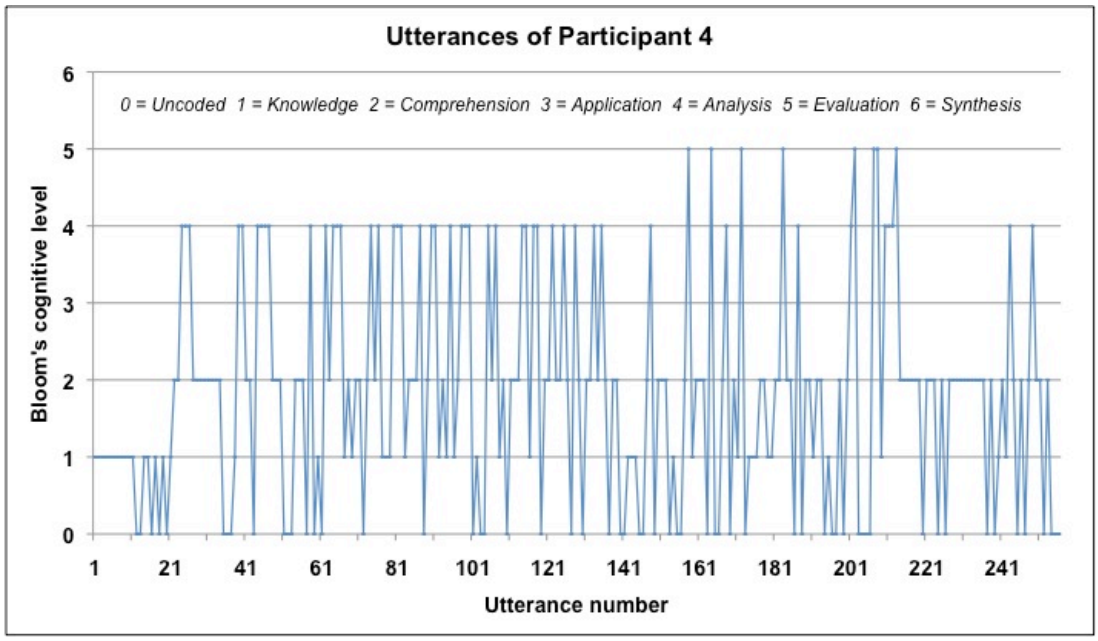


Figure 6-7. Participant 4's utterances, Bloom's cognitive levels in occurrence order.

The study required participants to map a sequence diagram to the underlying code. Participants did not need to identify defects, nor add functionality to the code. The results indicate that participants did not operate at the Synthesis level or at the Application level during the study (as noted earlier, there was only a single utterance at the Application level constituting less than 0.27% of a single participant's total number of utterances).

In this study's context, mapping a sequence diagram scenario to the underlying code, it was expected that the participants would neither operate at the Application nor the Synthesis level. This was because the task did not require them to perform any operation where they would have to operate at either of those two levels.

In Bloom's Taxonomy, the Synthesis level requires the creation of something new; while the Application level requires implementing, modifying or evolving some section of code. Hence, no participant functioned at those cognitive levels.

Table 6-3. Correctness of participants' solutions.

Participant	% of model solution covered	% of mis-matches
p1	42%	74%
p2	85%	37%
p3	33%	83%
p4	63%	25%
p5	62%	20%
p6	73%	27%

The participants' correctness of solution when compared to the model solution is displayed in Table 6-3. The results indicate a great variety in the participants' solutions when compared to the model solution. Participants' correctness varies from 33%-85%. Participants also incorrectly mapped between 20% and 83% of the executing lines of code with the model solution.

6.4 Discussion

Examining the expressed cognitive levels in conjunction with mapping the sequence diagram to the underlying code, the demonstration of both the Knowledge and Comprehension cognitive levels did not correlate with producing the correct solution. As participants were unfamiliar with the sequence diagram and corresponding code, this may have been a factor that led to the majority of their time being spent at the Knowledge and Understanding levels in attempting to execute the given task.

A top-down cognitive process strategy is generally required to map a sequence diagram to the underlying code. This is because the sequence diagram is at a higher abstraction level than the executing code, and the participant must map from the

higher level abstraction, the depicted interacting objects, down to the low level code that executes the scenario shown in the sequence diagram.

In order to successfully carry out this task, the participant needed to be able to identify relationships between different entities within the system, determining what they do, how they interact with each other and what methods are required at what times. As stated earlier, the performance of these tasks corresponds to the Analysis level of Bloom's taxonomy. Table 6-3 shows that participants were minimally successful in performing the given task, mapping the sequence diagram to the underlying code. Table 6-2 shows that, on average, only 19% of participants' utterances were at the Analysis level.

In this experiment, the results show that participants were unsuccessful in the given task and they also did not function for the vast majority of the time at the Analysis level. It is not possible to conclusively state that, had the inspectors operated more at the Analysis level, a greater number of their solutions would have been correct. Results from this experiment do indicate that operating at the Knowledge and Comprehension levels for approximately 70% of the time did not correlate to successfully completing the task.

Participants were completely unfamiliar with the system in question. It appears first and foremost that they were familiarising themselves with the system, indicated by the large number of utterances at the Knowledge and Comprehension levels of thinking.

At the Knowledge and Comprehension levels of thinking, one is usually familiarising oneself with the system at hand. At the Knowledge level, the participant is reading and recalling the code. At the Comprehension level, the participant is starting to describe the code in different terms; they are no longer simply reading it but, in one sense, they are rewording it.

Several issues arose from the data analysis of all ten participants. One was that no participant successfully identified the correct starting point of the scenario in comparison to the model solution. This caused problems for the participants because, in order to identify whether or not a line of code executed, one must first have identified all previous lines of execution. If previous lines were not successfully identified, this posed a challenge for the participant in that s/he needed to know whether or not the line s/he was currently looking at would be executed.

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

From identifying this, a simple recommendation can be made that within the code itself, a comment should be used to mark the location where a scenario starts

Another issue identified was the incorrect identification of polymorphism. Participants generally referred to the parent class when examining the code, even

though the descendant class was shown on the sequence diagram. Functioning at the Analysis level, breaking the material into its constituent parts would have meant that the developer understood the class's wider role within the program.

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

6.5 Conclusion

The nature of the process of mapping a sequence diagram to the underlying code, correlates well with the modified version of the Context-Aware Analysis Schema (Kelly, & Buckley 2006) for Bloom's Taxonomy's Analysis level. The Analysis level requires "breaking the material into its constituent parts" and identifying object interaction and communication within the system. A sequence diagram also requires the user to map from high-level functionality down to the low-level code. With these assumptions regarding the sequence diagram, it was expected that participants would have functioned at the Analysis level. However, this was not observed in this experiment.

In this experiment, it has been shown that the use of sequence diagrams to increase developers' system cognition with no prior system knowledge, is an ineffective way to build the developer's understanding of the system and its code. Had the inspectors had prior experience with the system, the task of performing the sequence diagram to

code mapping, may have enabled them to operate at higher cognitive levels for longer and more sustained time periods for the duration of the task.

Participants might have benefitted from beginning the task with a generic familiarisation methodology, as this would have enabled them to build a foundational system understanding prior to performing the sequence diagram mapping exercise. Separating the tasks, the initial acquisition of basic system knowledge and understanding from the sequence diagram mapping exercise, might have enabled participants to operate at the higher cognitive levels such as Analysis for longer and more sustained time periods.

Therefore, in creating methodologies and technologies to enable developers to operate at the higher cognitive levels, it is important that the tasks provide the developers with ways to accumulate the required lower cognitive levels and then move them along into the higher levels. Presenting developers with a task such as the one described here, neither assisted them to operate at the higher levels, nor did it assist them to successfully carry out the required task.

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

diagram to the code.

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

In the next chapter, software inspections will be examined to identify whether there is a correlation between the number of defects an inspector detects and the number of successful modifications s/he makes, unrelated to the detected defects, to the code base. A correlation between these two will support this dissertation, to use software inspections in the non-traditional area of increasing the cognitive levels and understanding of software systems.

Bibliography

Adelson, B. & Soloway, E., 1985, The Role of Domain Experience in Software Design, *Transactions on Software Engineering*, SE-11(11), pp. 1351-60.

Anderson, J.R., Farrell, R. & Sauers, R., 1984, Learning to Program in LISP, *Cognitive Science*, 8, pp. 87 - 129.

Bergantz, D. & Hassell, J., 1991, Information relationships in PROLOG programs: how do programmers comprehend functionality? *Int. J. Man-Mach. Stud.*, 35(3), pp. 313-28.

Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J. & Houston, K., 2007, *Object-oriented analysis and design with applications,* Third ed. Addison-Wesley Professional, Boston, USA.

Detienne, F., 1991, Reasoning from a schema and from an analog in software code reuse, *Empirical Studies of Programmers: Fourth workshop*, ESP91, p. 5.

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

Kelly, T. & Buckley, J., 2006, 14th IEEE International Conference on Program Comprehension ICPC '06, *A Context-Aware Analysis Scheme for Bloom's Taxonomy.* pp. 275-84.

Kelly, T. & Buckley, J., 2009, IEEE 17th International Conference on Program Comprehension ICPC '09, *An in-vivo study of the cognitive levels employed by programmers during software maintenance.* pp. 95-9.

Letovsky, S., 1986, Empirical studies of programmers: First workshop, *Cognitive processes in program comprehension.* pp. 58 - 79.

Page-Jones, M., 2000, *Fundamentals of Object-Oriented Design in UML,* Addison-Wesley Professional.

1956, *Taxonomy of Educational Objectives Cognitive Domain,* Bloom, B.S. ed. David McKay Company, Inc..

7.0 Chapter Seven

Relationship between Defect Detection and Modifications

7.1 Introduction

The literature review and problem definition chapters identified that performing a software inspection requires the inspector to understand the code s/he is examining. As discussed in Chapter Four, comprehending something means understanding it, and understanding means perceiving the intended meaning. A successful software inspection results in either no defects being detected in the system, or defects being detected which can subsequently be corrected. In order for these outcomes to occur, the inspector perceives the code's intended meaning and identifies whether or not the code successfully carries out its intended meaning.

Chapter Five compared three code inspection techniques and reported the results from when industry professionals executed the code inspection and compared them with the results of inspections by student participants who carried out the same task. The results indicated that none of the three inspection techniques out-performed another, or under-performed in detecting defects, defects due to delocalisation or in generating more false positives.

Based on the finding from Chapter Five, that is, that a particular inspection technique did not influence the number of defects detected, this research now examines the application of inspection techniques for non-traditional roles within the software development process. This phase of the research sought to address the research issues of identifying if there was a correlative relationship between the number of defects one detects while carrying out a software inspection, and the number of modifications one can make to the code base after completing the code inspection.

This phase of the research reported in this chapter was broken down into several parts. The goals were to:

1. investigate if there was a relationship between the number of defects a participant discovered and the number of subsequent modifications they successfully made (modifications were not related to the defects);
2. determine if participants approach the task of code modification in significantly different ways if they have previously performed a code inspection of that code;
3. examine if participants perceived that an inspection assisted them in making subsequent modifications (adding functionality) to the inspected code; and
4. observe the way in which participants performed a code modification: those who performed an inspection compared with those who did not perform an inspection.

The experiment in this phase of the research consisted of two participant groupings. One group of participants performed a Checklist-Based Reading (CBR) code inspection. The Checklist-Based Reading technique was chosen for this section of the research because it is considered the de facto software inspection technique and the technique most widely used within the software development industry (Laitenberger, & DeBaud 2000; Tyran, & George 2002).

Upon completing the code inspection, participants were then required to add new functionality to the code. The new functionality was not specifically related to the defects they had detected.

The second group of participants was required to carry out the task of adding the new functionality to the code base without previously carrying out an inspection of the code.

7.1.1 Hypothesis

It was expected that the participants who had carried out the software inspection would go about making the changes in a way that more resembled one of the two sample solutions. That is, they walked through the code more systematically because they had previously seen the code and also had some understanding of the location of items within the code.

Hence, a hypothesis was created for this section of the study. The null hypothesis was stated as:

H_0: Participants who inspected the code prior to making changes to it, more

systematically (as described in the two sample solutions) go about adding new functionality.

The alternate hypothesis was:

H_1: Participants, who have not inspected the code prior to making changes to it, go about adding new functionality in a less systematic way (as described in the two sample solutions).

In order to test this hypothesis, a two-way t-test was carried out. This test is used to compare the responses in two groups, where the responses of one group are independent of the responses of the other group (Moore, & McCabe 1999).

7.2 Methodology

7.2.1 The software artefacts

The software artefacts used in this study were created as part of a third year software engineering course tutorial. The software was a navigational data recording system. It was to be used by people learning to navigate with a compass. Every 10 minutes, the software required users to input their current position that they had calculated by hand, using a map and their compass. Upon completing the navigation task, the software compared user-entered positions with the data collected from a GPS system. By comparing the two data sets, the navigation instructor is able to ascertain the accuracy with which the student navigator calculated his/her position. Each transportation mode being used was fitted with a GPS device, and if the student was walking, it was fitted to the backpack.

Figure 7-1 shows the class diagram for the inspected system. The Location class was the inspected class. The class stored the positions entered by the student navigators. The class was responsible for storing and giving access to past positions entered and enabled the distance between two recorded points to be calculated. The class contained 126 executable lines of code.

The modification that inspectors were asked to make following the inspection was to provide the class with the ability to store the user's altitudinal position and then include this when calculating the distance between points.

Calculating the user's altitude as well as the 10 minute timing was not within the scope of the navigation software system.

The software inspectors were presented with 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; and
6. Questionnaire.

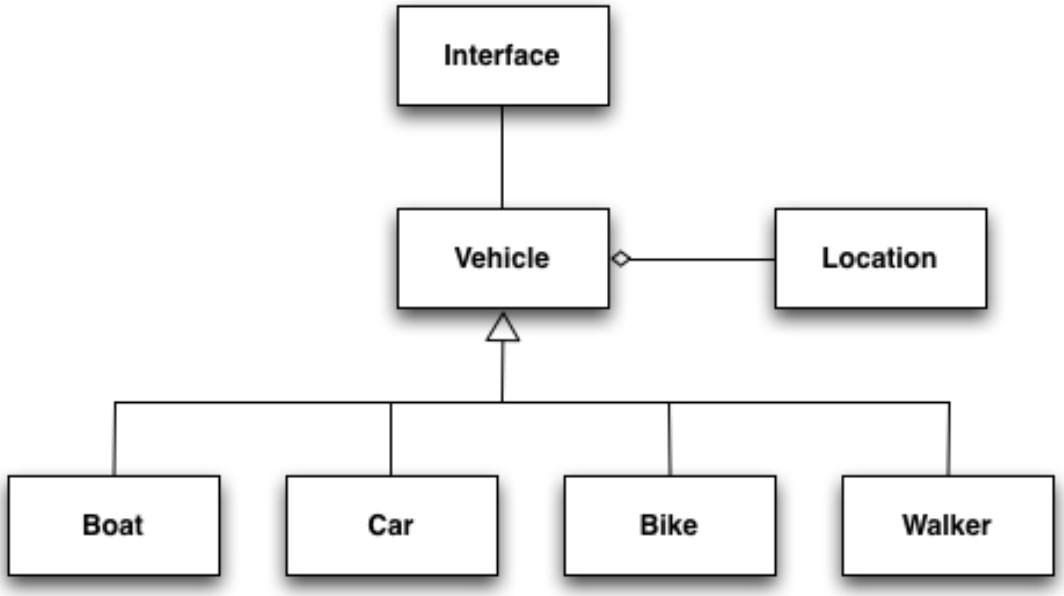


Figure 7-1. Class diagram of the inspected navigation system.

Since a software inspection is a static process, the code was not executed during the inspection process. Participants were informed that the code compiled, executed and produced results but because of a software inspection's static nature, participants were asked not to compile and execute the code for themselves. The artefacts used within this study were paper-based, and participants were also given access to the Java API online documentation.

7.2.2 The participants

The participants were all final year undergraduate students enrolled in one of the following three Bachelor degree programs: Computer Science, Information Technology or Software Engineering. The base level requirements established for all participants to take part in this study was to have successfully passed two introductory Java programming courses and the first two Software Engineering courses within their degree programs. These courses are common to all three degree

programs in which the participants were enrolled.

The establishment of these criteria for participation created a consistent experience baseline. All students had successfully completed these requirements, thereby providing a common experience level. With this common experience level established, an assumption could then be made that participants all had a similar base knowledge. There was still the possibility of differing experience levels within the student grouping and the impact that this could have on the study. This has been addressed in the Threats to Validity section in Chapter Four.

The first participant group was presented with the listed software artefacts and were required to perform a code inspection. Immediately following the code inspection, the participants were informed of the nature of the code modification that was required and carried it out.

The second participant group was also presented with the software artefacts, except they were not given the checklist or defect reporting sheet. The participants in this group were informed of the nature of the code modification that was required and carried it out without having previously conducted a code inspection.

For the purposes of the experiment, results from the first participant group were used to test if there was any relationship between the number of defects an inspector detects and the number of modifications that an inspector makes to the inspected code base. Results from the second participant group were used to compare the ways in which participants went about making code modifications if they had previously

carried out a code inspection, with the way in which participants went about making code modifications having not previously performed a code inspection.

7.2.3 Carrying out the study

The experiment consisted of two separate stages. The first stage took place during a third year software engineering course tutorial. The participants were required to execute a Checklist-Based Reading code inspection on the Location class Java file. The experiment's second stage required participants to add new functionality to the Location class Java file they had just inspected.

Participation in the experiment's second stage was voluntary and was not part of the software engineering course. No part of the material or learning done through the experiment was examinable as part of the participants' degree coursework. Upon completing the inspection tutorial, students were presented with the option to participate or not in the second stage. The experiment followed immediately after the tutorial ended and was conducted in the participant's own time. Participants were not paid to take part.

The tutorial was one hour long and segmented into different stages. In the first stage, a training exercise was conducted that introduced participants to how to carry out a Checklist-Based Reading inspection. Participants were then given 30 minutes to execute the code inspection. There was then 10 minutes in which participants worked with a partner to calculate their individual defect yield, their combined team yield and estimate how many defects still remained within the code. The final 10 minutes was devoted to a group discussion about their defects yields and how

successful they believed they had been in detecting defects.[4].

The inspection process was an individual task for which participants were requested not to interact with others carrying out the inspection. Participants were given 30 minutes to execute the inspection and were advised to read the natural language specification first and then examine the class diagram. As they detected defects, these were recorded in a defect recording sheet. Participants entered the line number of the defect in the class and wrote a brief description of the detected defect. Participants were not required to correct the defect; nor did they have to explain how the defect could be corrected. Their task was simply to identify and describe the defect, and nothing else.

7.2.4 Seeded defects

Twelve defects were seeded into the inspected class. For example, three defects 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 then was replicated in three different locations. The remaining defects were seeded based upon prior research (Dunsmore et al. 2000; Dunsmore et al. 2001; Dunsmore et al. 2002) and also errors familiar to the author while writing code.

[4] These final two sections of the tutorial were outside the scope of this research but are based on the Capture Recapture technique that Humphrey describes (Humphrey 2000).

7.2.5 The new functionality

The study's second section required participants to add new functionality to the inspected code. The new functionality requirement was explained to participants and they were given 30 minutes to add the needed code. As with the code inspection, adding the new functionality was an individual task and participants were asked not to interact with other participants during the process. Adding new functionality was conducted online and participants were able to compile their code as often as they chose.

The additional functionality to be implemented did not directly involve correcting the previously detected defects. The added functionality was to allow the user to also store their altitude. Participants were asked to ensure that their changes worked appropriately, produced correct results, and did not add new defects to the code. The seeded defects remained in the code and it was left to the participant to determine whether or not these needed to be corrected. Several participants asked if the defects needed to be corrected and the response given was that their changes to the code should ensure that the new functionality operated correctly.

A model solution for the new functionality was created by the researcher in consultation with three domain experts. This model solution required 37 modifications to be made to the code base.

Participants from both groups were given 30 minutes to perform this modification. During this time, a screenshot was automatically taken every five seconds capturing:

1. The code being viewed;

2. The changes to the code;
3. The order in which changes were made; and
4. The way in which the participant went about making those changes.

7.2.6 Data collection, analysis and results

The defect detection data was collected during this experiment by participants entering a defect description into a defect-recording sheet. This was compared with the known defect listing. During analysis of the collected data, false positives were identified.

The final code files created and/or modified by each participant were used to collect the modification data along with the automatic screen-shots taken while they were carrying out the modification task.

A demographic sheet and questionnaire, completed by each participant, were used for collecting qualitative data. The quantitative data was analysed using the R statistical software package as well as Statistical Package for the Social Sciences (SPSS).

Eighteen students participated in this study. The first group, which carried out the code inspection as well as the adding of new functionality, contained 11 participants. The second group, which only carried out the adding of new functionality, contained seven participants. In the first group, participant three's results were removed from the quantitative data analysis as they failed to detect a single defect and only reported modifications. Participants were required to carry out a code inspection

detecting defects, and the data indicated that this participant failed to carry out the task correctly. Consequently, the data for that participant was removed.

A server failure while capturing screen shots meant that a large portion of that data failed to be written to disk. Therefore, the screenshots of four participants were not available for analysis. However, the code files that included the changes and new functionality were still available.

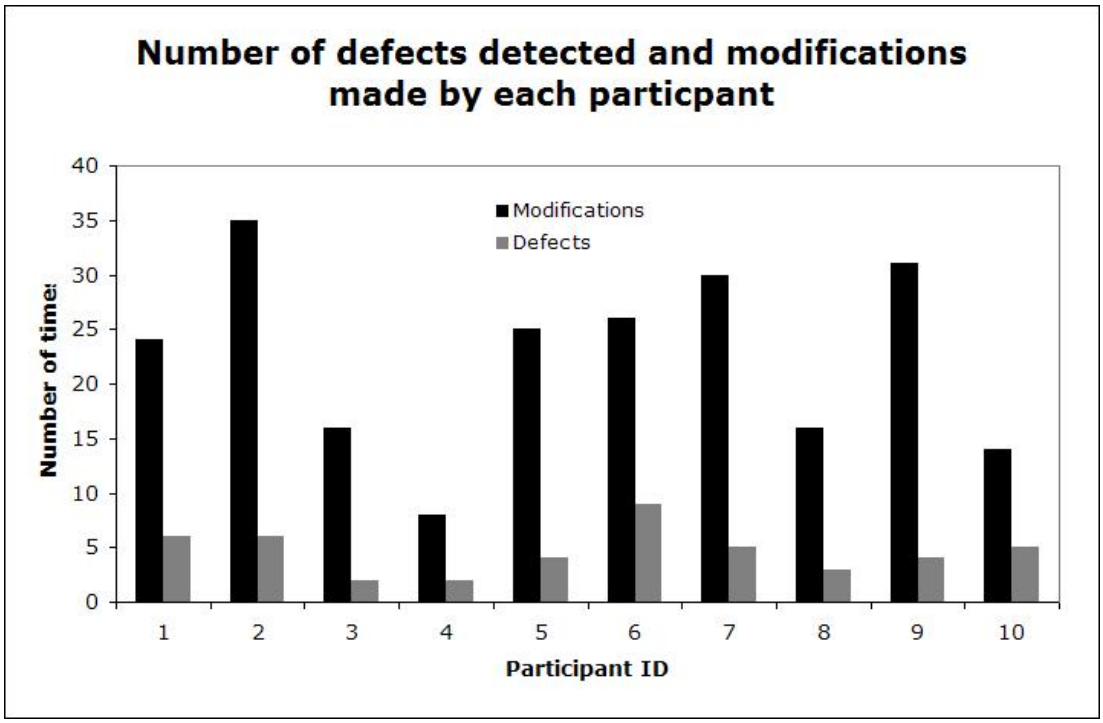


Figure 7-2. Number of defects detected and changes made by each participant.

7.3 Results

7.3.1 CBR inspection followed by addition of new functionality

The number of detected defects and changes made by participants in the first group is shown in Figure 7-2. When viewing this graph as well as other comparisons between the detected defects and changes made to the code, the difference between

the number of defects within the code and the number of changes that needed to be implemented must be noted: there are 37 changes and only 12 seeded defects.

The box plots in Figure 7-3 and Figure 7-4 show the distribution of the first group's defect detection and data for code changes. Both box plots indicate there are no outliers within the results and that the data appears to be normally distributed.

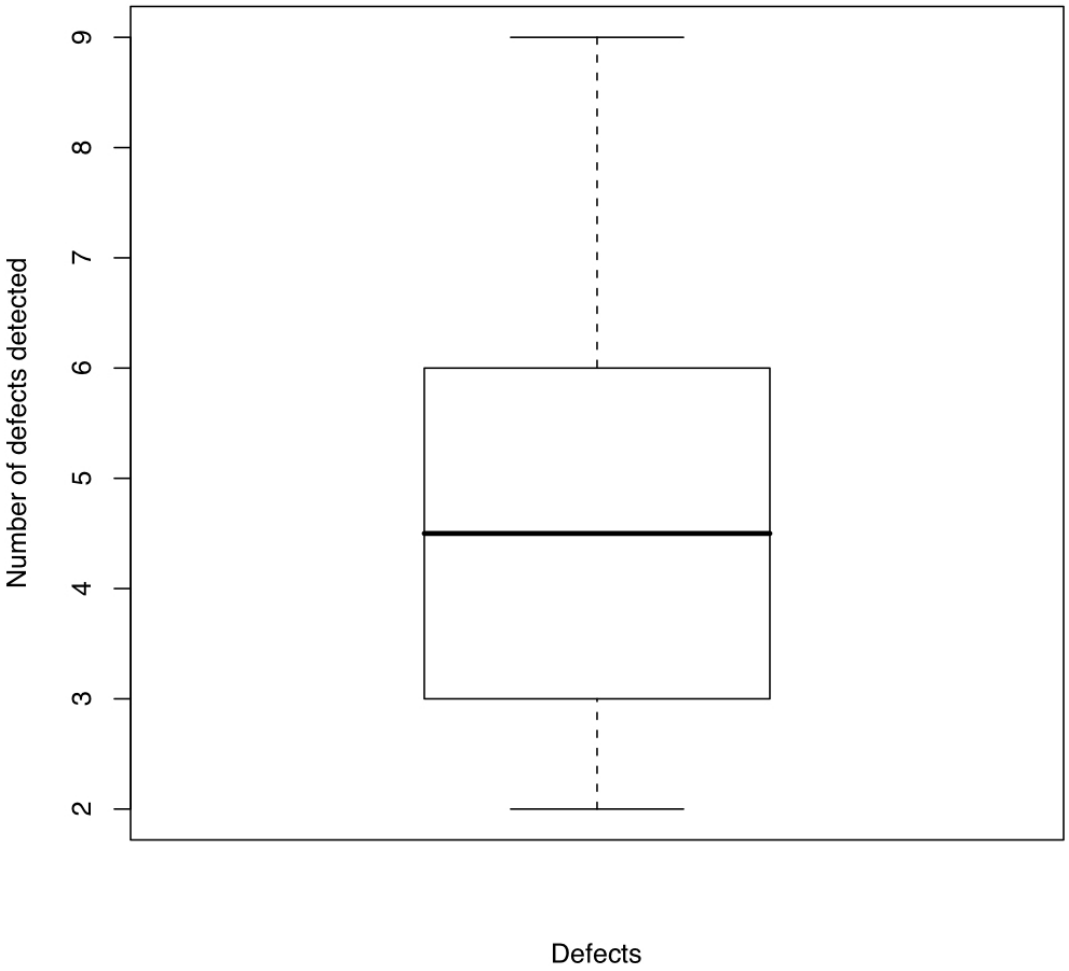


Figure 7-3. Box plot of the defects detected by participant group one.

The descriptive statistics for defect detection and code changes made by the first participant group are displayed in Table 7-1

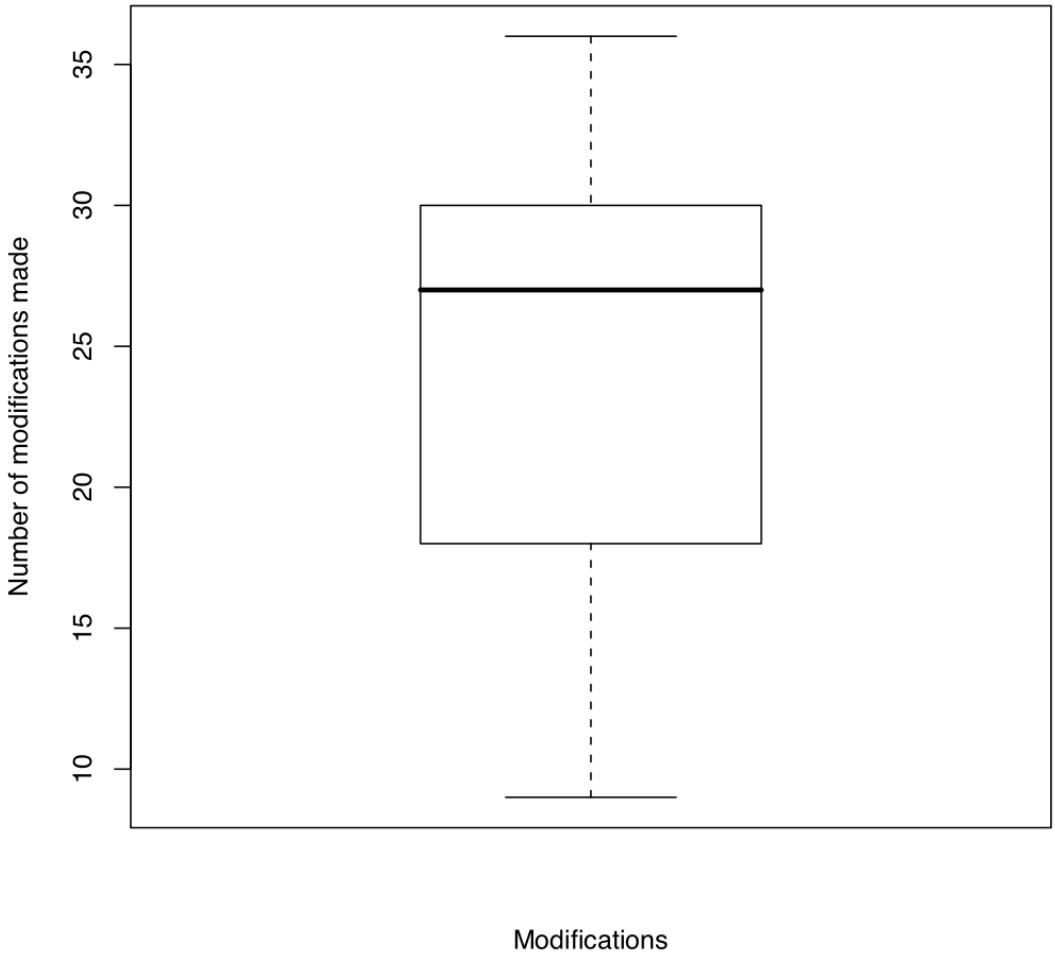


Figure 7-4. Box plot of the modifications made by participant group one.

Table 7-1. Descriptive statistics from group one.

	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

Following the collection of the defect and code change data, a regression analysis was conducted in order to determine if there was a correlation between these two data sets. Figure 7-5 shows the relationship between the number of defects detected by the inspection and the number of changes to the code that each participant made. The line of best fit indicates a weak correlation; as the number of detected defects

increases, the number of successful changes the inspector makes to the code also increases.

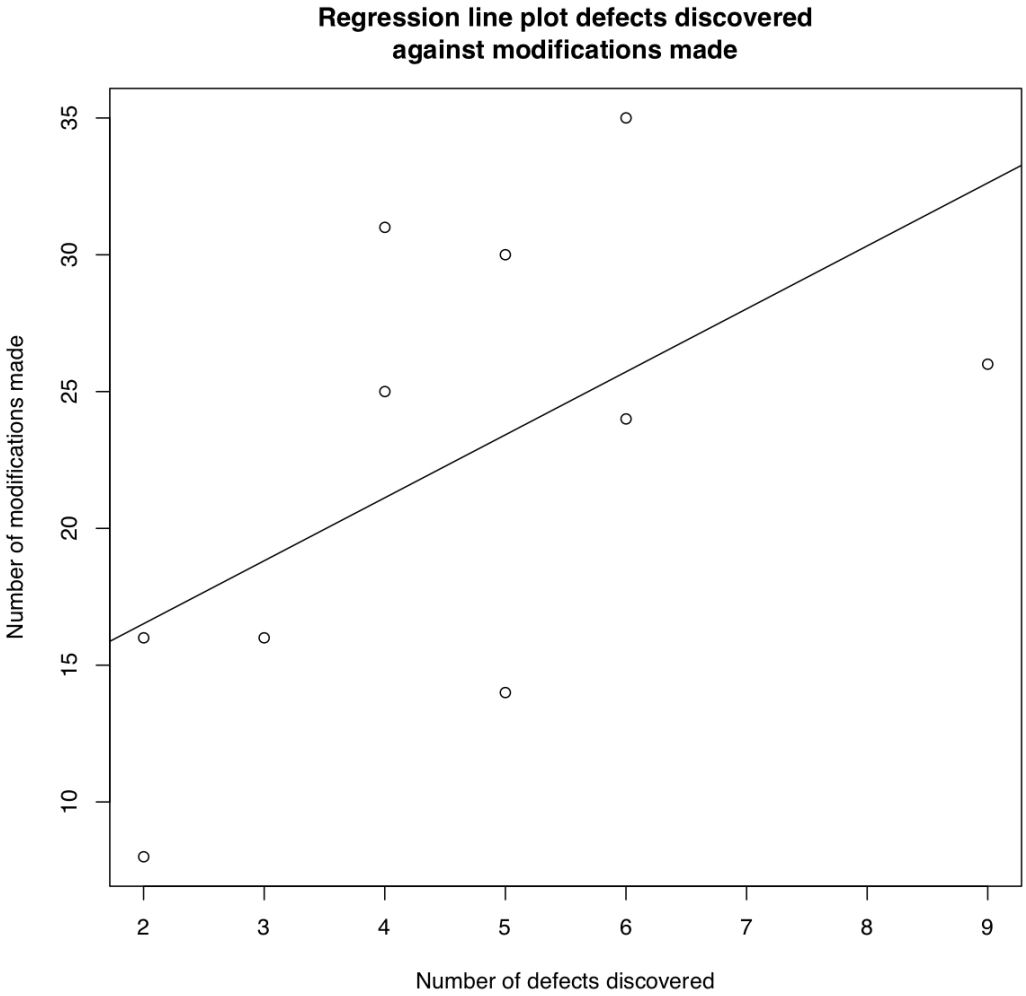


Figure 7-5. Scatter plot demonstrating the relationship between detecting defects and modifications made.

The linear regression analysis results showed R-squared = 0.32. The two values, the best fit and the R-squared values, indicate the existence of a weak correlation between the number of defects detected and the number of successful changes one makes to the code.

A Pearson Product-Moment correlation analysis of the data was also performed, r =

0.56 and p-value = 0.09. These results indicate a significant correlation between the number of detected defects and the number of successful changes made to the code, at the 90% confidence interval.

7.3.2 Defect and change types

The number of times each defect was detected during the inspection is shown in Figure 7-6. Defects one, five, nine and eleven had the highest detection rates. The code for defects nine, ten and eleven assigned an incorrect Boolean value when validating latitude, longitude and map number respectively. Defect one had incorrect parameter ordering, longitude and latitude were reversed and defect five was the failure to use a parameter within the method to which it was passed. It was expected that by using the checklist, these defects would be detected. This was because the checklist questions directly referred to these defect types.

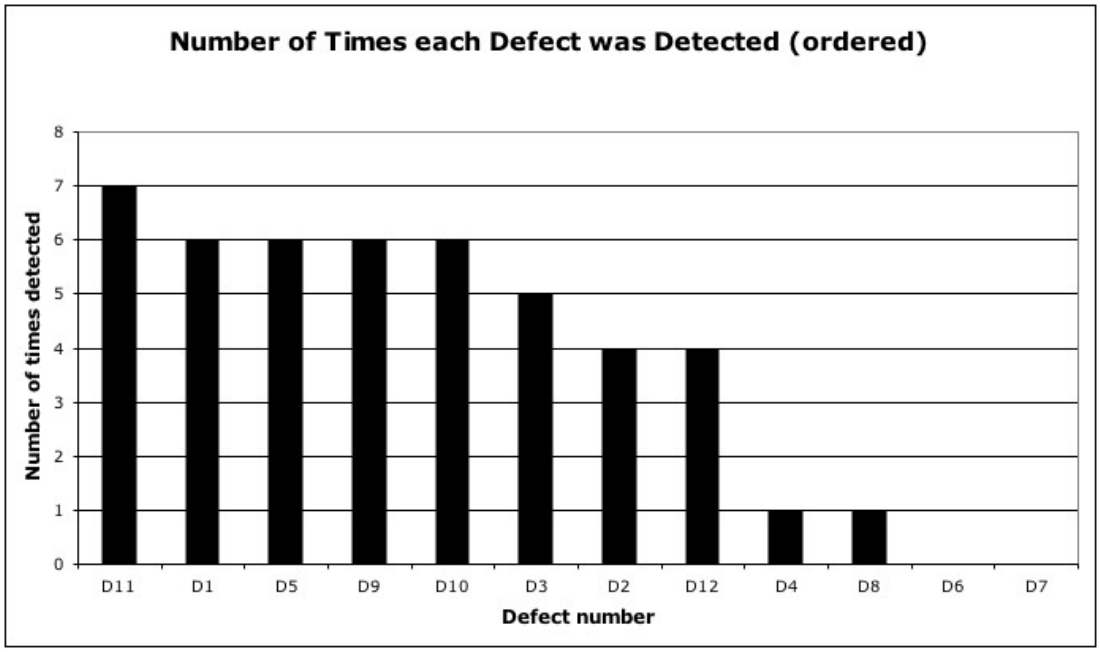


Figure 7-6. Number of times each defect was found (ordered from highest to lowest).

Figure 7-6 shows that defects six and seven were not detected at all throughout the duration of the inspection. In defect six '==' was used in comparing doubles for equality. Defect seven used the same structure the '==' to compare two Vector objects for equality.

The checklist contained no direct question that related to these defect types. From examining the data, in order for these defect types to have been detected, a question that directly asked about this error type would have been needed. However, checklists are there to guide the inspector as they search the code for defects. Checklists cannot be expected to ask every possible question to catch every possible defect type as this would make the checklist almost limitless in length as well as cause it to violate the recommendation that it be, at maximum, one single-sided approximately A4 size page (Brykczynski 1999).

From Figure 7-6, it can be seen that defect four was detected only once. The code appeared to return a Vector object, but in Java the return value is actually 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.

In order to understand these defective uses of Vector class objects, the inspector is actually required to understand the Vector class provided by Java. If participants did not understand the Vector class, it was expected that they would go to the Java API documentation and examine that in order to better understand that class. However,

no participant examined the Java API documentation regarding the Vector class.

Figure 7-7 displays the number of times each of the required changes was made to the code base. All participants made changes one, two and three. These changes required the additions of the class field altitude and the class constants for its maximum and minimum values.

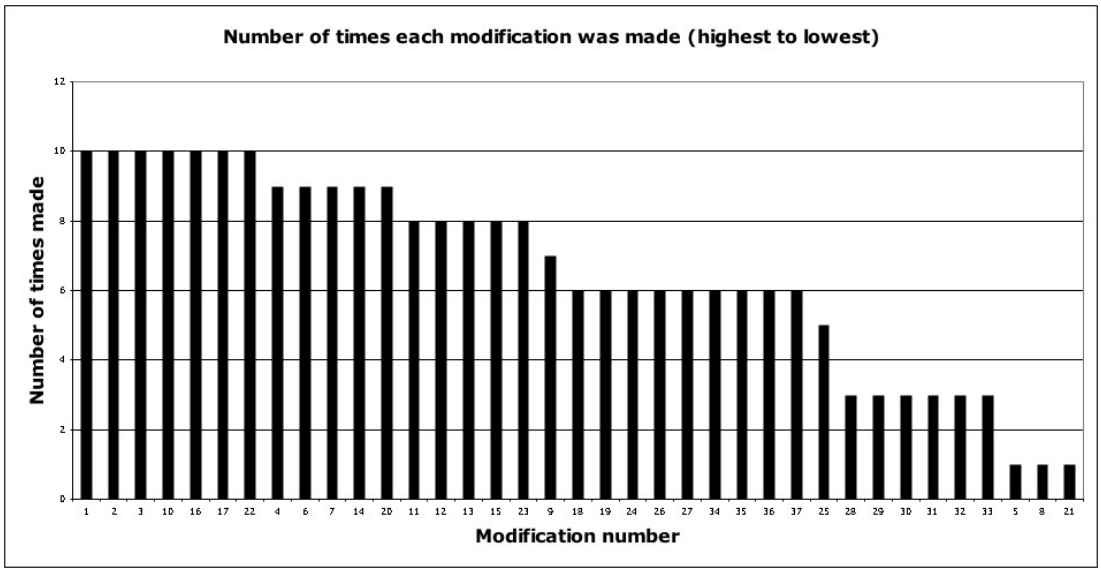


Figure 7-7.Number of times each modification was made (ordered from highest to lowest).

Change one was considered essential, adding the new class field, since, if this was not carried out, the other changes were deemed as unneeded and hence the new functionality would not be added. Changes two and three were not always the second and third modification performed, but on several occasions they were implemented when participants realized they were needed.

Changes five, eight and 21 were made the least number of times by participants. Changes five and eight were located in two different constructor calls. Each call

required modifying the initial Vector size, enabling it to increase by the correct increment sizes if needed.

Change 21 required creating a tolerance value to enable the accurate comparison between doubles. It is interesting to note that this change was not implemented in the code and also, defect six, comparing two doubles using the "==" operator, was also not detected by any of the inspectors.

These two occurrences highlight the literature findings that checklists within an organisation need to be an evolving artefact (Brykczynski 1999). Here, a defect within the code has not been detected and it has now also been replicated in another location within the class that was inspected.

7.3.3 Analysis of participants' perceptions

Once participants had completed the code changes required for the new functionality or their 30 minutes had finished, they were asked a series of questions regarding the inspection and changes. The answers to these questions provided qualitative data related to the participants' perceptions regarding the impact that the inspection had upon their ability to make the required changes to 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 or 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 the first question, all participants answered ‘yes’. Participant three, who had not reported any detected defects also answered ‘yes’ to this question. Participant three detected no defects during the inspection process and ‘yet’ in his opinion, the inspection still better prepared him to modify the code than had he not performed the inspection. Participant three’s answer indicated the possibility that code inspections may assist in improving inspectors’ understanding, even if it is not successful in assisting them to detect defects.

To the second question, nine participants answered ‘yes’ while one participant answered ‘no’. When justifying their responses 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 types of statements indicated that the participants perceived that they had a better grasp and understanding of the code that needed to be changed and added to because of the knowledge they had acquired while executing the code inspection.

With question three, seven participants strongly agreed, while two participants agreed and one participant failed to answer the question.

The answers to these questions indicate that participants believed that carrying out an inspection prior to implementing changes in the code to give the system new

functionality, assisted them in their understanding and comprehension of the system, therefore enabling them to add the newly requested functionality and to add other new functionality to the system by adding more classes.

7.3.4 The changes order

The screen shots taken every five seconds while participants carried out the task of adding new functionality to the code depicted the way in which participants went about performing the required changes.

The author, in collaboration with two other experts, created two sample solutions. These sample solutions indicated how the participant would go about making the changes to the code to add the new functionality (these were not the solutions to the defects within the code or the changes that were to be made).

The two sample solutions started with the same three assumptions. First, in order to add the new functionality, a class field representing the altitude value that was now to be recorded, needed to be present first. The second was that the constants used to constrain the maximum and minimum values of the class field were created second. The third was to initialise the new class field in the constructors. After these three base assumptions, the two sample solutions varied.

Sample solution one then continued to step through the class in a top-to-bottom manner. That is, the inspector was assumed to make the changes by moving down the code line by line. When code was encountered that needed to be changed, it was changed.

Sample solution two stepped through the code making the required changes on an as-encountered method. For example, when the participant was changing method A, and method A made a call to method B, the participant would immediately move to method B. In method B, the required change(s) were implemented immediately and then the participant returned to method A. However, while implementing changes inside method B, if method B made a call to method C then the inspector, in the same pattern as described, would follow the call to method C, implementing any needed changes there and then return to the calling method, in this case method B. This pattern continued until all required changes were completed.

From each sample solution, an order of the needed changes within the code was established. The two ideal order paths through the code in order to make the needed changes were then compared against the changes order paths that each participant performed.

Table 7-2. The average R^2 values for the way in which each group modified the code.

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

Participant group one consisted of those who had carried out the software inspection prior to adding the new functionality. Participant group two consisted of those who had not carried out the software inspection but were immediately required to carry out the task to add the new functionality. Table 7-2 shows the average R^2 values for the way in which first and second group went about implementing the changes to the code. In both cases, group one who had performed the inspection prior to making the

changes, had better R^2 values; that is, they were closer to one than was the group who had not performed the software inspection.

A two-way t-test was performed that compared the way in which the inspectors from each of the two participants groups implemented the changes in the code. In this instance, the paths that the participants from each group took through the code changes, were compared to the first sample solution, the top to the bottom of the class. A p-value of 0.001 was returned which indicates a significant difference in how the code was changed between the two groups. Participants who had previously performed the code inspection, made their changes to the code in a more systematic manner, one that more closely reflected sample solution one, than the participants who had not performed the prior inspection.

A two-way t-test was performed that compared the way in which the inspectors from each of the two participants groups implemented the changes in the code. In this instance, the paths that the participants from each group took to make the code changes were compared with the second sample solution, the as-it-is-encountered within the class. A p-value of 0.02 was returned which indicates a significant difference in the ways that each group changed the code. Participants who had previously performed the code inspection made their changes to the code in a more systematic manner, one that more closely reflected sample solution two, than the participants who had not performed the prior inspection.

Based on these results for this experiment conducted under the described conditions, the null hypothesis $H_{0,}$ can be accepted: Participants, who inspected the code prior to

making changes to it, they more systematically (as described in the two sample solutions) go about adding new functionality.

These results indicate that inspectors who had previously performed a code inspection, made changes to the code in a more systematic manner. Their solutions more closely resembled the two model solutions created than did the solutions of those who had not previously carried out a code inspection. Participants who had not performed the inspection previously worked through the code but in a more Ad hoc manner. This could be seen in the screen shots showing that they moved up and down the class where they appeared to be looking for the needed code. This may be attributed to the fact that the developers who had performed the inspection knew the code structure as well as the location of methods that needed to be changed. Already in possession of this knowledge, they moved in a more direct manner to the code areas needing changes than did those participants who had not performed the inspection. The latter group therefore also used their time to search the code looking for locations where the changes needed 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.

The results indicate that inspectors strongly believe that performing a code inspection prior to changing code improves their code understanding as well as their ability to add new functionality. The statistical analysis results indicated that there is a significant influence at the 90% confidence interval between the number of defects detected by an inspector and the number of successful changes they are able to make

(n = 18). By performing an inspection prior to changing the code by adding new functionality, developers more systematically and directly applied the needed changes to the code.

7.4 Conclusion

Based on the results from this experiment and the context in which it was conducted, it can be deducted that carrying out code inspections is an effective method for increasing developer productivity. This is because, as the developer already familiar with the code and how it works, the time needed to carry out the changes to the code base in which they are working, may well be reduced.

The results from this study demonstrate that the undertaking of a CBR inspection positively affects a developer's ability to make changes to the code. After performing a CBR inspection, developers made changes to the code more systematically than did 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.

Chapter Five showed that there was no significant difference in detecting defects between three different software inspection techniques. This chapter has shown that a correlation exists between the number of defects an inspector detects and the number of successful changes that the inspector makes to the same code base following a code inspection.

In the next chapter, several code inspection techniques will be examined, focusing

on the different cognitive levels at which inspectors operate while conducting a software inspection using the different inspection techniques.

Bibliography

Brykczynski, B., 1999, A survey of software inspection checklists, *SIGSOFT Software Engineering Notes*, 24(1), p. 82.

Dunsmore, A., Roper, M. & Wood, M., 2000, Proceedings of the 22nd international conference on Software engineering ICSE '00, *Object-oriented inspection in the face of delocalisation.* Limerick, Ireland, pp. 467-76.

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

Dunsmore, A., Roper, M. & Wood, M., 2002, Proceedings of the 24th International Conference on Software Engineering ICSE '02, *Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection.* pp. 135-44.

Humphrey, W.S., 2000, *Introduction to the team software process,* Addison--Wesley, Massachusetts.

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

Moore, D.S. & McCabe, G.P., 1999, *Introduction to the practice of statistics,* third ed. W.H. Freeman and Company, New York.

Tyran, C.K. & George, J.F., 2002, Improving software inspections with group process support, *Commun. ACM*, 45(9), pp. 87-92.

8.0 Chapter Eight

Inspectors' Cognitive Levels during a Code Inspection

8.1 Introduction

This chapter examines the cognitive levels that developers demonstrated while carrying out a code inspection searching for defects within the code being inspected. Chapter Five showed that there was no significant difference between inspection techniques with respect to the defect detection rate. Research results reported in Chapter Seven indicated that software inspections assist developers in three different areas: 1) the number of defects detected correlates with the number of successful additions to the code, 2) a code inspection improves an inspector's understanding of the code, and 3) the way in which developers make changes to the code is more structured and systematic if they have performed a previous code inspection prior to making modifications.

The previous chapter reported that carrying out an exercise to map a Usage-Based scenario in a sequence diagram to the underlying code executing the scenario did not require developers to operate at the higher cognitive levels within Bloom's Taxonomy. Also, the task did not enable developers to produce a correct mapping

between the sequence diagram and the underlying code.

These two results suggest that developers need a certain level of system understanding prior to carrying out such a task. Unless they have acquired this prior knowledge, developers have a more difficult time functioning at higher cognitive levels when carrying out the task.

Building upon these results, this chapter examines developers' expressed cognitive levels, categorised using Bloom's Taxonomy. The assigned task is to carry out an inspection to detect defects. However, because of the results already reported, this experiment focuses on identifying the different cognitive levels developers express while carrying out the inspection task, not the number of defects detected. This chapter sought to identify the cognitive levels expressed by novice student inspectors as they carried out a software code inspection using differing code inspection techniques.

8.2 Methodology

In this experiment, three different participant groups were established to implement a code inspection using one of three different inspection techniques. Each member of the three groups carried out an individual code inspection in which they implemented either the Ad hoc, Checklist-Based or Abstraction-Driven Reading technique. For the duration of the inspection, participants were required to "think-aloud" (Ericsson, & Simon 1993).

A single Java class from within a larger software system was chosen for the code

inspection.

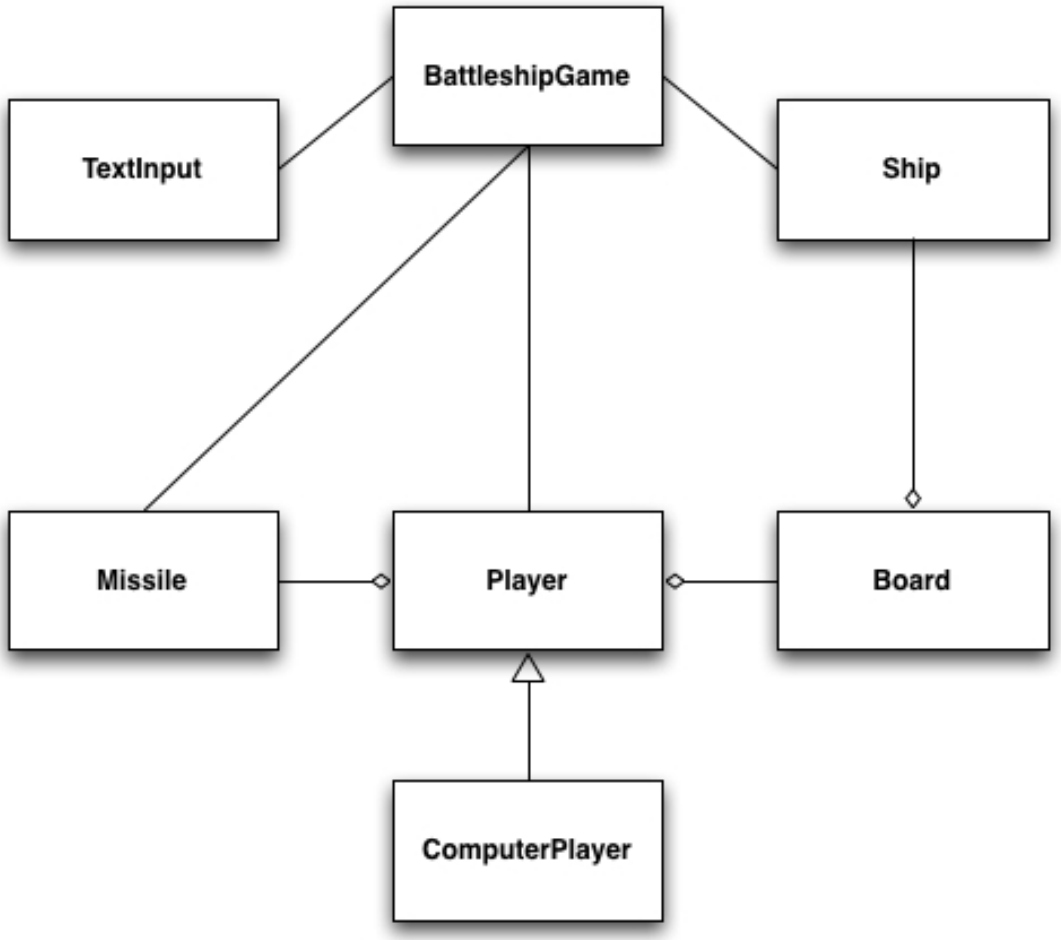


Figure 8-1. The Class diagram of the Battleship game created for this experiment.

8.2.1 The software artefacts

The software artefacts used in this experiment were created specifically for this experiment. The software system was a text-based version of the Battleship Game. Figure 8-1 is the Battleship software system's Class diagram. The game was designed for a single player to challenge the computer. Participants were required to inspect the Board class Java file. This class represented the game board and was responsible for storing ships' positions, receiving attacks, determining if a ship had been hit and/or sunk and also determining if the game had been won.

Participants were asked not to compile and execute the code for themselves. However, they were informed that the code did compile and execute. The class contained 169 effective lines of code and had 13 seeded defects.

Participants were given access to 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); and
6. A defect-recording sheet.

All the artefacts were online except for the defect-recording sheet and the checklist. The participants were issued with paper versions of both of these artefacts. The online Java API documentation was also available for participants to refer to if needed or desired.

8.2.2 The participants

The participants in this experiment were student developers. The students were enrolled in one of three Bachelor Degree programs: Computer Science, Information Technology or Software Engineering. The experiment did not form any part of their formal course work; it was completely voluntary and students took part in their own time. The base level requirements established for students to participate in the experiment were that they be in the third or fourth year of their degree, have

successfully passed the two Java introductory units and the first two Software Engineering units. One participant had recently graduated and just enrolled in a computing-based PhD.

A baseline for participants' experience level was established by setting the above-mentioned criteria as a requirement in order to participate.

To attract participants, the experiment was advertised throughout the computing degree courses on campus. All students participated in their own time and were not paid for participating. Participation in the experiment was voluntary and students were informed that no part of the material or learning that took place through the experiment was part of any course or examinable within participants' degree programs. Participation would have no influence on their marks in any subject within their degree program.

Prior to taking part in the experiment, all participants had attended a two-hour lecture in which software inspections were discussed. The lecture included information such as the historical basis of inspections, empirical research results and different software inspection techniques discussed within the literature.

Before commencing the inspection, all participants took part in a short training session, the purpose of which was to familiarise participants with the specific inspection technique they were to implement during the experiment. They also performed two training exercises from Ericsson and Simon (Ericsson, & Simon 1993) to practise the "think-aloud" protocol.

8.2.3 Seeded defects

Thirteen defects were seeded within the inspected code in the manner described in the Seeded Defects section within Chapter Four. 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 copying and pasting a similar code section and then failing to make the changes from its original form into the form required in its new location within the class. This defect led to the system failing because it permitted the same location to be bombed multiple times instead of informing the player that that location had already been attacked.

8.2.4 Carrying out the experiment

The participants were randomly assigned to the different inspection technique groups. An arbitrary order of assigning participants was established: Ad hoc, Checklist-Based and Abstraction-Driven Reading. This meant that the first person to enter the room to participate was assigned to the Ad hoc method. The second person to enter was assigned to the Checklist-Based Reading method and the third person was assigned to the Abstraction-Driven Reading technique. This process was then repeated until each group consisted of five members.

The participants were then given 30 minutes to carry out a code inspection on the Board class using the inspection technique that they had been assigned. The inspection process was an individual task and participants were asked not communicate with anyone else participating within the experiment.

The experiment was conducted over several days, hence, participants were asked not to discuss the experiment with other students in order to keep the participant knowledge level the same when they came to participate.

For the duration of the inspection, participants were required to "think-aloud." If participants were quiet for approximately 30 seconds, they were then prompted by the researcher to continue to think-aloud. The think-aloud data was recorded via a microphone fitted in the headset of participants.

A screen shot was automatically taken every five seconds in a background process. The screenshots provided an image of what was being displayed on the screen. This information allowed the following information to be viewed:

1. The code currently being viewed; and
2. The way in which participants went about executing the inspection task.

Software was created that compared the epoch time, in seconds, between sequential screen shots. It then created the same number of screen shots, from the first image in the sequential comparison, as was the time difference between the two compared screen shots so that a pseudo video could be created from the inspection process.

The audio from the think-aloud data was then added to the pseudo video. This provided a visual perspective of what participants were doing when they were verbalising their thoughts and actions.

8.2.5 Threats to validity

The threats to validity this experiment was subjected to are discussed in the Threats to Validity section in Chapter Four.

8.3 Results

The think-aloud recordings were transcribed and broken down into sentences, with each sentence considered an utterance. Using the modified version of the Context-Aware Schema, two researchers independently coded the same set of 130 utterances using the Context-Aware Schema. Differences in interpreting the schema were discussed to clarify the schema's interpretation and application. The Cohen's Kappa statistic was used to determine the inter-observer reliability between researchers' utterance categorisations. The Cohen's Kappa was 0.605, which is considered acceptable, and the author then coded the remaining utterances.

Table 8-1 Utterance example by category.

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

Table 8-1 gives a coded example of an utterance from each category of Bloom's Taxonomy, coded using the modified version of the Context-Aware Schema. The

column labelled Graph Number is the corresponding Y-axis value on Figure 8-8, Figure 8-9 and Figure 8-10. The "Uncoded" category accounts for utterances that could not be categorised in this manner. An utterance was placed into the Uncoded category for one of two reasons: first, if it was inaudible and therefore not possible to transcribe and second, if it did not relate to the inspector's current cognitive process. For example, a participant talking on his mobile phone about what he was going to do later that evening was categorised as Uncoded.

Table 8-2. Inspection utterance summary.

Participant ID	Knowledge	Comprehension	Application	Analysis	Evaluation	Synthesis
p1 Ad hoc	54%	24%	1%	10%	11%	0%
p3 Ad hoc	77%	11%	0%	6%	7%	0%
p9 Ad hoc	51%	12%	1%	13%	23%	0%
p10 Ad hoc	53%	31%	0%	9%	7%	0%
p11 Ad hoc	62%	17%	0%	10%	12%	0%
Mean Ad hoc	**59%**	**19%**	**0%**	**9%**	**12%**	**0%**
p2 CBR	37%	21%	0%	12%	30%	0%
p4 CBR	39%	27%	0%	10%	24%	0%
p5 CBR	21%	12%	0%	15%	52%	0%
p7 CBR	23%	22%	0%	9%	46%	0%
p8 CBR	26%	19%	0%	12%	42%	0%
Mean CBR	**27%**	**20%**	**0%**	**11%**	**41%**	**0%**
p6 ADR	43%	28%	0%	10%	19%	0%
p12 ADR	52%	27%	1%	5%	16%	0%
p13 ADR	44%	27%	1%	11%	17%	0%
p14 ADR	49%	18%	5%	9%	19%	0%
p17 ADR	60%	12%	0%	5%	19%	0%
Mean ADR	**50%**	**22%**	**2%**	**8%**	**22%**	**0%**

Table 8-2 shows participants' utterances broken down into their Bloom's Taxonomy classification. The Uncoded utterances have been removed from the listing and also

have not been included when calculating the percentages.

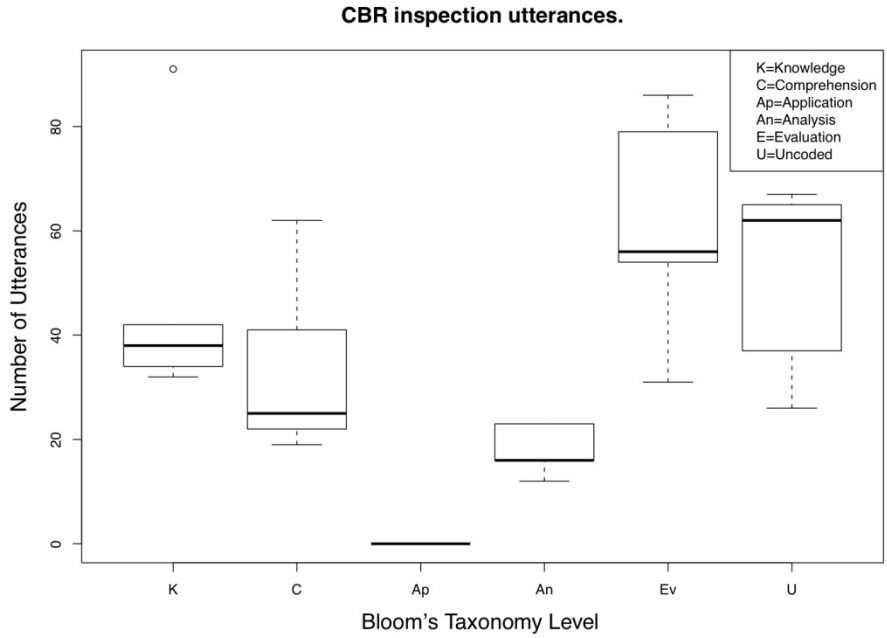


Figure 8-2. Distribution of utterances in Bloom's Taxonomy from those performing a CBR inspection.

The box plots shown in Figure 8-2, Figure 8-3 and Figure 8-4 display the distribution of each utterance category for the three different code inspection techniques. For both the Abstraction-Driven and Ad hoc Reading techniques, the Knowledge category is high when compared to the CBR technique (it is noted that the CBR technique contains one outlier). The CBR technique has a much greater distribution within the Evaluation category than either of the ADR and Ad hoc reading techniques. In all three techniques, the Uncoded category also has a large distribution.

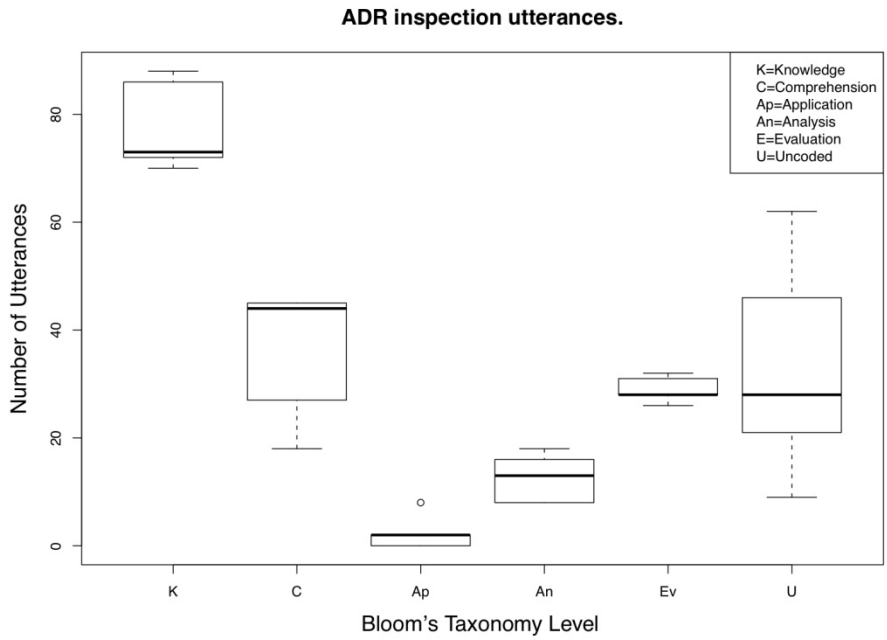


Figure 8-3. Distribution of utterances in Bloom's Taxonomy from those performing an ADR inspection.

Figure 8-5, Figure 8-6 and Figure 8-7 display the individual inspector's utterances for the duration of the inspection, with the Uncoded utterances removed.

The Ad hoc inspection utterances graph in Figure 8-5 shows that, on average, 60% of the participants' utterances were at the knowledge level, which is the lowest cognitive level in Bloom's Taxonomy. Of the three tested inspection techniques, this is the highest average operating at the Knowledge level. These same participants 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 code inspection.

Figure 8-4. Distribution of utterances in Bloom's Taxonomy from those performing an Ad hoc inspection.

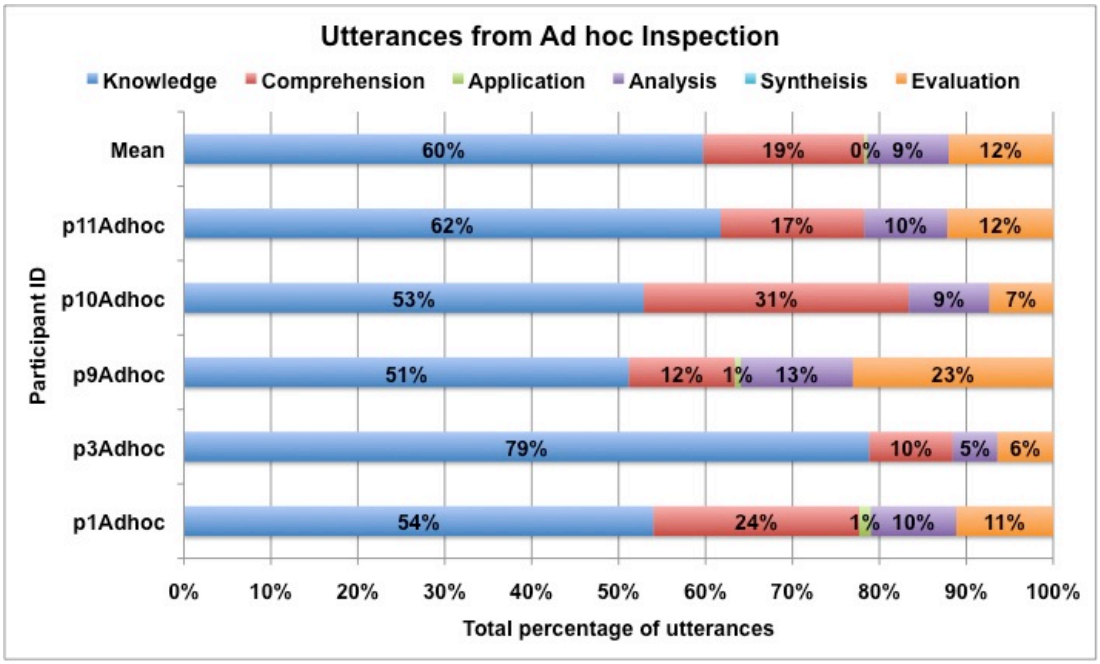


Figure 8-5. Breakdown of each participant's utterances from the Ad hoc inspection into Bloom's Taxonomy.

The ADR inspection utterances are given in Figure 8-6, which shows that

participants in that group operated at the Knowledge level 50% of the time on average. Participants 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, on average, a higher utterance count at the Evaluation level of Bloom's Taxonomy than did those who carried out the Ad hoc inspection. The Ad hoc and ADR inspectors showed by their utterances that, on average, they were operating at the Comprehension cognitive level for approximately 20% of the time.

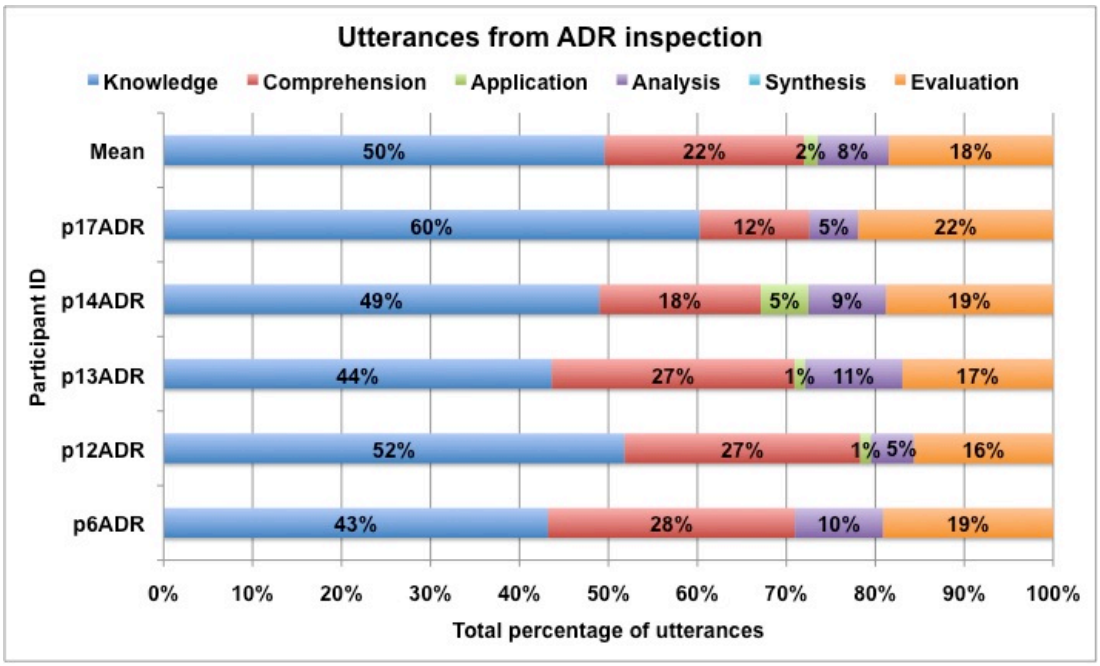


Figure 8-6. Breakdown of each participant's utterances from the ADR inspection into Bloom's Taxonomy.

The Knowledge level is the level at which the inspector is reading through the code. At this level for example, the inspector reads through the code and is simply repeating what s/he has read. At the Comprehension level, which accounted for about 20% of utterances for the Ad hoc and ADR techniques, the inspector is reading the code and starts to describe it in terms different from the way it is written.

Hence, both the Ad hoc and ADR inspection techniques required inspectors to operate at the lower cognitive levels within Bloom's Taxonomy, Knowledge and Comprehension.

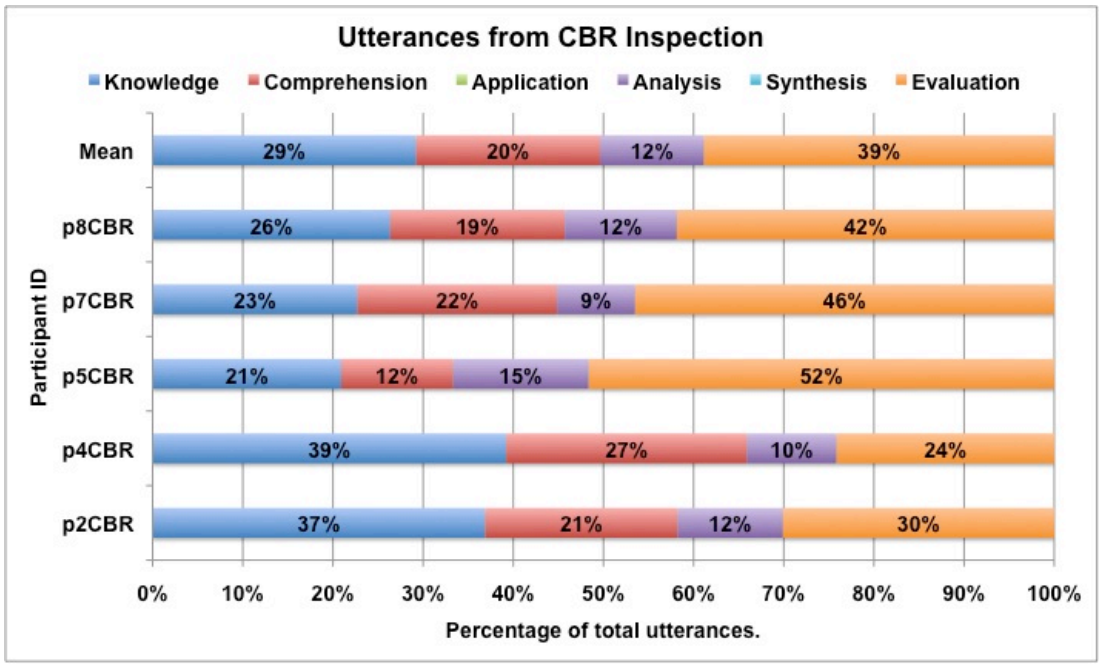


Figure 8-7. Break down of each participant's utterances from the CBR inspection according to Bloom's Taxonomy.

The bar graph in Figure 8-7 displays the utterances for the individuals who carried out the CBR inspection. On average, about 29% of their utterances were in the Knowledge category, and 20% were in the Comprehension category. For the Knowledge category, this is considerably lower than for those who carried out the Ad hoc and ADR inspections. The Comprehension level for CBR inspectors is on par with both the Ad hoc and ADR inspection techniques.

Figure 8-7 shows that, on average, 39% of their utterances were at the Evaluation level of Bloom's Taxonomy, with one CBR participant having 52% of his utterances

at the Evaluation cognitive level.

Figure 8-5, Figure 8-6 and Figure 8-7 show that for the student inspector, the CBR inspection technique facilitated operation at the Evaluation level more consistently than both the Ad hoc and ADR inspection techniques.

In a software inspection, the inspector is searching the code for defects, and making judgements regarding the code's correctness. The Evaluation level is where judgements are made and hence, it is an appropriate level of thinking for the inspector. The inspector needs to know the intended meaning of the code and evaluate it to ascertain whether the intended purpose has been achieved. If the code is not fulfilling its intended purpose, then a defect within the code has been detected. Therefore, the facilitation of the Evaluation cognitive level is appropriate.

A Kruskal-Wallis statistical analysis test was carried out that compared each category in Bloom's Taxonomy, with that same category, for the three different inspection techniques. For the categories Comprehension, Application, Analysis and Synthesis, the results indicated that there was no significant statistical difference between these categories amongst the three different inspection techniques. For the categories Knowledge and Evaluation, the test returned p-values of 0.004 and 0.006 respectively. This indicates a significant difference between the CBR technique and both Ad hoc and the ADR techniques in these two areas.

Figure 8-8, Figure 8-9 and Figure 8-10 show the cognitive levels at which the developers were operating sequentially. One participant graph from each inspection

technique was chosen for display.

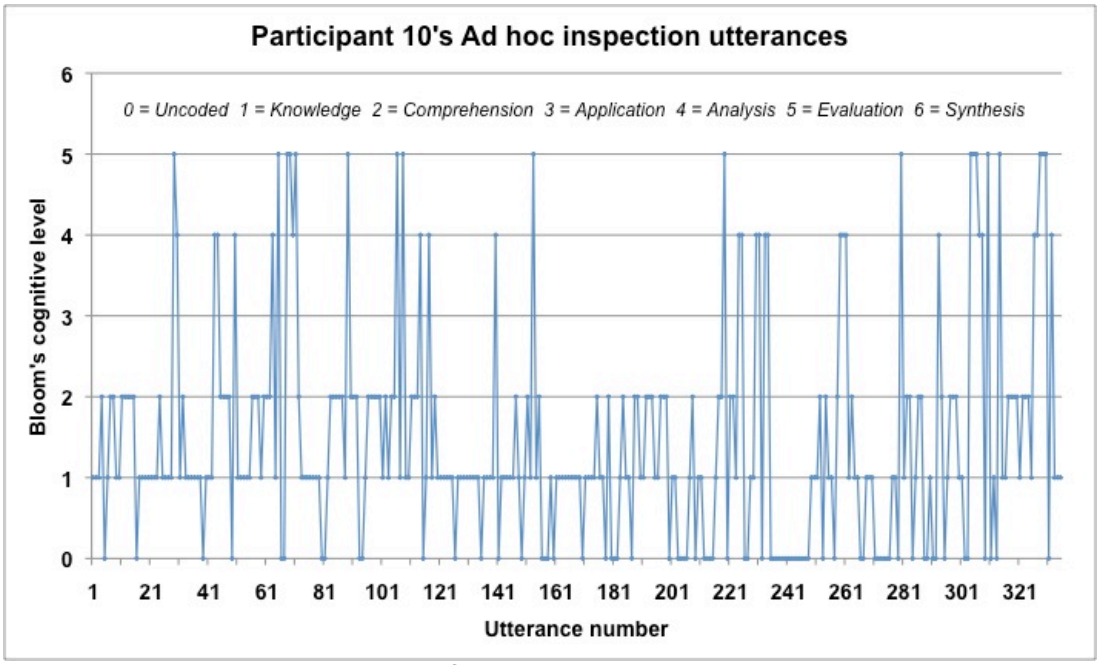


Figure 8-8. Participant 10's order of utterances categorised using Bloom's Taxonomy.

Figure 8-8 shows that participant 10, who carried out the Ad hoc inspection, operated at the lower cognitive levels of Knowledge and Comprehension for a large portion of time. He intermittently moved from the Knowledge and Comprehension levels up to the Analysis and Evaluation levels and then returned to the Knowledge and Comprehension levels. However, for the vast majority of the inspection, he functioned at the lower cognitive levels.

Figure 8-9 shows that participant 13, who carried out the ADR inspection, started out at the Knowledge and Comprehension levels but then moved up to the Analysis and Evaluation levels. He tended to function at those higher cognitive levels for certain periods, and then returned to the lower cognitive levels. Compared with the inspector who carried out the Ad hoc inspection, he did operate more consistently at the Analysis and Evaluation cognitive levels.

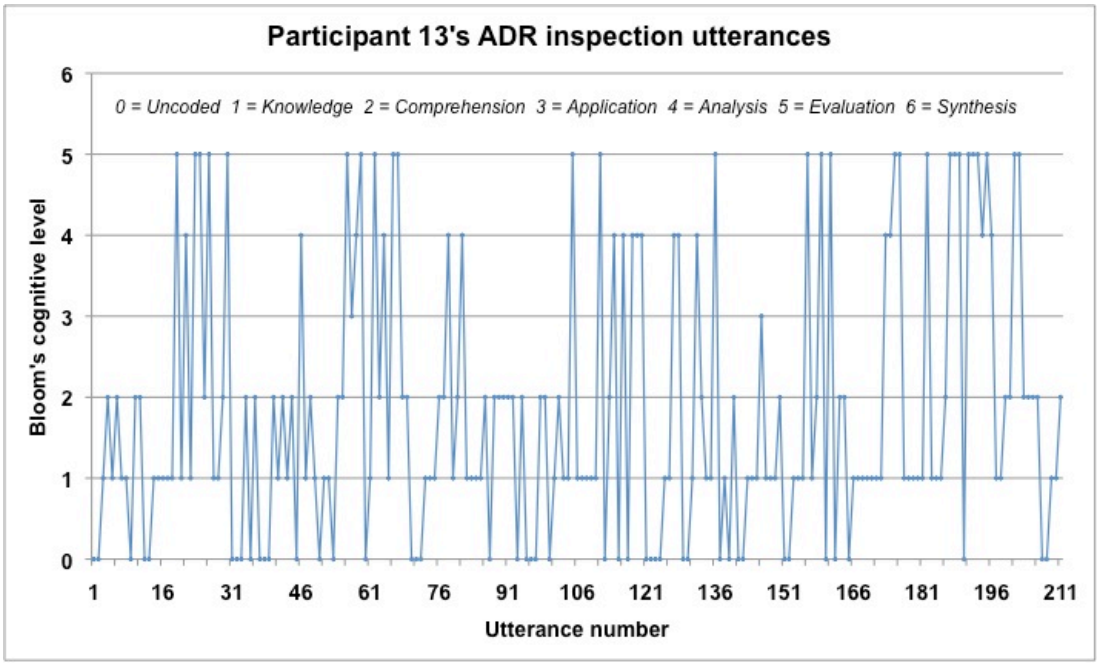


Figure 8-9 Participant 13's order of utterances categorised using Bloom's Taxonomy.

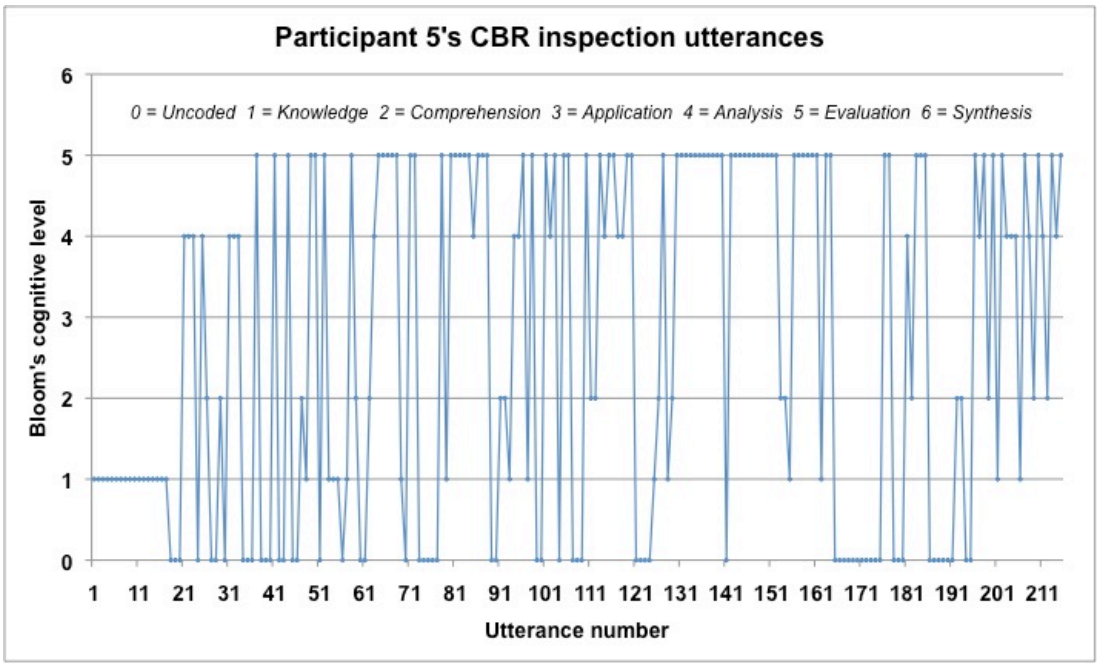


Figure 8-10. Participant 5's order of utterances categorised using Bloom's Taxonomy.

Figure 8-10 shows that participant five, who carried out the CBR inspection, commenced the inspection at the Knowledge level and then progressed to the Analysis and Evaluation cognitive levels. There were several points when he moved

back down to the Comprehension and also Knowledge levels, but overall he more consistently operated at the higher cognitive levels.

8.4 Discussion

8.4.1 Ad hoc

The Ad hoc code inspection technique is the least structured technique examined in this thesis. There are no instructions or directions regarding how the inspection is to be carried out. The inspection technique relies solely on the inspector's previous experience. The inspector is required to execute the inspection and determine what, if any, defects exist within the code.

In this case, student inspectors were given the software artefacts and asked to conduct an inspection and determine if the code contained defects. The Ad hoc method gives no further instructions regarding how to do this. Hence, the student inspectors needed to rely on their own past experiences to carry out the task with no additional guidance.

Examining the categorised utterances, the student inspectors' utterances were mostly in the lower cognitive levels. These two cognitive levels are still essential to the overall task of understanding something, but in the case of carrying out a 30-minute code inspection, the Ad hoc technique did not facilitate the higher cognitive functions when implemented by student participants.

8.4.2 ADR

The ADR technique required the inspector to write a short natural language abstraction, describing functionality contained in the method and class. This inspection technique provides more structure regarding how to carry out the inspection task than does the Ad hoc inspection technique. The technique is still general and doesn't tell the inspector how to perform the inspection. Hence, the student inspectors implementing this technique had to work out for themselves how to go about the inspection.

The ADR inspection technique requires that a natural language abstraction be written describing the method, as well as a list of defects. Summarising and describing the code suggests that there would be more utterances at the Comprehension level when compared to the Ad hoc inspection technique.

However, this experiment did not show this to be the case. Similar to the Ad hoc inspection technique, the inspectors tended to operate mostly at the Knowledge level. Inspectors tended to operate at the Knowledge level, recalling and repeating the code they had read.

In the Board class there were a total of 15 methods that needed to have an abstraction written. No inspector wrote all 15. One inspector managed to write 10 abstractions while no other inspector came close to this number of abstractions. The abstraction writing falls into the Comprehension level. However, there simply weren't enough abstractions written to significantly increase the Comprehension cognitive level utterance count.

For this inspection technique to have a major impact upon the Comprehension utterance count, more time may be needed for the inspection. Thirty minutes may not have been enough time to be able to write the required number of abstractions.

8.4.3 CBR

The CBR inspection technique is very structured, explicitly describing the process to be implemented. The technique describes what to do and the steps needed to implement it, whereas the Ad hoc and ADR techniques leave the inspector to decide much of how to go about the inspection. Inspectors answer ‘yes’ or ‘no’ to questions and in doing so decide whether or not a defect exists within the code.

Figure 8-2, Figure 8-7 and Figure 8-10 show that for the CBR inspection, the Evaluation level comprised the majority of the inspectors’ utterances; on average 39% of all coded utterances were at the Evaluation cognitive level. Whereas the Ad hoc and ADR techniques facilitated the Knowledge and Comprehension cognitive levels, the CBR technique tended to facilitate inspectors functioning at the Evaluation cognitive level.

As the CBR technique requires a ‘yes’ or ‘no’ answer to a question, correctly implementing this technique facilitates the inspector to function at the Evaluation cognitive level. This is because answering the checklist questions requires the inspector to evaluate whether or not the code correctly implements what the question asks.

Figure 8-7 shows that participants p2CBR and p4CBR did not operate at the Evaluation cognitive level to the same extent as did participants p5CBR, P7CBR and p8CBR. These two participants, p2CBR and p4CBR, operated largely at the Knowledge cognitive level during the inspection. In the think-aloud data, both of these participants stated that they were deviating from the CBR inspection to inspect as they saw fit. However, the other three participants systematically followed the CBR inspection process. These results indicate that the deviation from the CBR technique may have been the reason that those two inspectors did not operate at the Evaluation cognitive level to the same extent as did the other three inspectors.

8.5 Conclusion

Figure 8-5, Figure 8-6 and Figure 8-7, along with the statistical analysis, highlight that the CBR inspection technique will help inspectors to operate at the Evaluation level in Bloom's Taxonomy. According to these results, the CBR technique is significantly more advantageous than the Ad hoc and ADR inspection techniques for aiding the student inspector to operate at the higher cognitive levels.

Operating at this higher cognitive level is important for several reasons. First, operating at the Evaluation level enables the developer to inspect the code and identify what the code does and whether it fulfils the intended task. Second, when they do make changes, they will make changes that either work or will introduce new defects in another unforseen location.

The following chapter examines the second part to this experiment. Upon completing the code inspection, participants were given a change request to the code

and required to carry that out.

Bibliography

Ericsson, K.A. & Simon, H.A., 1993, *Protocol Analysis,* The MIT Press.

9.0 Chapter Nine

Developers' Cognitive Levels While Adding New Functionality

9.1 Introduction

This chapter examines the cognitive levels that developers demonstrated while carrying out the task of adding new functionality to a code base. The earlier chapters have demonstrated significant differences between novice/student and industry professional software developers/inspectors as they carry out software inspections but no significant difference with respect to inspection techniques used. Experience is a greater contributing factor to the success of the inspection than is the inspection technique used. A correlation between the numbers of defects one detects and the successful number of changes to the inspected code base has also been demonstrated. The previous chapter demonstrated a difference between inspectors' expressed cognitive levels while carrying out a software inspection task using different inspection techniques. The results indicated that different code inspection techniques facilitated developers operating at different cognitive levels throughout the inspection process.

This chapter builds on that experiment, by examining developers' cognitive levels as

they carry out the task to add new functionality to the code base they have just finished inspecting.

This experiment investigated developers' cognitive levels as they carried out the task of adding new functionality to the code base they had just finished inspecting for defects. The purpose of the experiment was to determine whether there were differences in the inspectors' expressed cognitive levels as a result of their using different inspection techniques when conducting the code inspection. The experiment also examined the expressed cognitive levels of student participants presented with this same code base, requiring them to add the same new functionality to the code base but without previously inspected the code.

Participants, who took part in the code inspection reported in the previous chapter, were asked to add new functionality to that code base immediately following the inspection. Five more student developers also took part in the exercise, but they had not previously carried out the code inspection.

The research in this chapter involved:

1. Categorising, analysing and evaluating the expressed cognitive levels;
2. Examining the expressed cognitive levels in relationship to the code inspection technique implemented during the previous code inspection; and
3. Examining the cognitive levels at which a developer operates while carrying out a code modification when they have not previously inspected the code base for defects.

9.2 Methodology

This experiment consisted of four participant groups. The first three were the same groupings who undertook the code inspection reported in the previous chapter. However, a fourth participant grouping was introduced that consisted of five new student participants. The new participants also met the minimum requirements criteria, set out in the previous chapter. None had previously performed the code inspection.

The participants were required to think-aloud as they carried out the modification task. Screen shots were also captured every five seconds. The five new participants were trained in how to think-aloud prior to adding the functionality. This group was given the same training exercises as those used with the original groups.

9.2.1 The software artefacts

The text interface implementation of the Battleship game was again used for this experiment. The five new participants were presented with the same software artefacts as the original 15. However, the Java code provided contained no defects, and this was also given to the original 15 participants. All participants were also given two additional artefacts: the natural language specification and diagram describing the new functionality to be added to the code base.

9.2.2 Carrying out the experiment

All participants, including those who had not carried out the software inspection, were given 30 minutes to add the newly requested functionality. After 30 minutes,

participants were required to stop, regardless of whether or not they had completed the task. During the process of adding the new functionality, participants were required to think-aloud and a screen shot was taken every five seconds, as in the case of the inspection.

The functionality was to be added to the Board class. The functionality was to give players the ability to place ships in a diagonal down to the right manner, as shown in Figure 9-1.

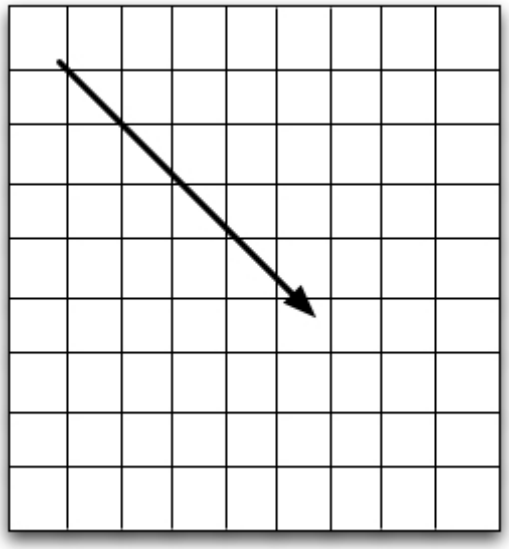


Figure 9-1. The modification should allow for boats to be placed on the board in this manner.

9.2.3 Threats to Validity

The threats to validity that this experiment encountered are discussed in the Threats to Validity section in Chapter Four.

9.3 Results

An utterance example from each category within Bloom's Taxonomy, as well as an Uncoded utterance, from the experiment is displayed in Table 9-1. The seventh category (Graph Number 0) shown in Table 9-1 is Uncoded, and does not form part of Bloom's taxonomy. As has been previously noted, utterances in this category were either unintelligible or unrelated to the task at hand, such as talking on the mobile phone during the experiment.

Table 9-1. Example of utterances.

Graph Number	Bloom's Level	Utterance Example
0	Uncoded	I can't type
1	Knowledge	Fleet counter equals new ship inShip
2	Comprehension	so to place a ship you need to check that there is enough space
3	Application	place ship okay that's the only place it's used
4	Analysis	this is externally controlled
5	Evaluation	I'm pretty sure that will be okay
6	Synthesis	we need another method there else set Ocean diagonal

The total percentage of each participant's utterances in each category is displayed in Table 9-2. The Uncoded utterances have been omitted from the listing and have not being included in the calculation for the percentage breakdown.

The box plots in Figure 9-2, Figure 9-3, Figure 9-4 and Figure 9-5 show the distribution of the utterances, including the Uncoded utterances. The data is grouped into each box plot according to which inspection technique the participant carried out prior to adding the new functionality. The distribution indicates that the spread of utterances varies not only within Bloom's Taxonomy categorisation but also between the different inspection techniques.

Table 9-2. Modification utterance summary.

Participant ID	Knowledge	Comprehension	Application	Analysis	Evaluation	Synthesis
p1 Ad hoc	36%	11%	49%	0%	4%	0%
p3 Ad hoc	60%	3%	31%	0%	6%	1%
p9 Ad hoc	33%	7%	47%	3%	10%	0%
p10 Ad hoc	24%	12%	37%	13%	12%	3%
p11 Ad hoc	34%	5%	52%	3%	2%	3%
Mean Ad hoc	**37%**	**7%**	**43%**	**4%**	**7%**	**1%**
p2 CBR	33%	6%	33%	5%	7%	15%
p4 CBR	34%	14%	23%	10%	14%	5%
p5 CBR	23%	5%	43%	5%	13%	11%
p7 CBR	30%	14%	32%	6%	10%	8%
p8 CBR	14%	7%	47%	9%	9%	13%
Mean CBR	**27%**	**9%**	**36%**	**7%**	**11%**	**11%**
p6 ADR	19%	8%	35%	11%	13%	14%
p12 ADR	4%	4%	47%	8%	21%	1%
p13 ADR	36%	7%	29%	12%	15%	1%
p14 ADR	32%	4%	50%	5%	7%	1%
p17 ADR	25%	5%	47%	2%	18%	2%
Mean ADR	**23%**	**6%**	**42%**	**8%**	**15%**	**4%**
p12 none	48%	5%	27%	11%	9%	0%
p16 none	53%	5%	29%	1%	11%	2%
p18 none	39%	10%	30%	11%	8%	2%
p19 none	44%	19%	18%	15%	3%	2%
p20 none	43%	17%	21%	5%	14%	1%
Mean none	**45%**	**11%**	**25%**	**8%**	**9%**	**1%**

Inspectors' utterances for the duration of the task are shown in Figure 9-6 to Figure 9-9. The Uncoded category has been omitted.

Figure 9-6 displays the utterance breakdown for the participants who had not performed a code inspection prior to this task. The majority of utterances from these

participants are at the Knowledge cognitive level. The next largest utterance category is at the Application level. These participants had not seen the code prior to carrying out the task to add new functionality. Hence, they needed to invest time to acquire enough knowledge about the system to successfully carry out the task. The Knowledge level is the lowest level, and is consistent with just reading through the code.

Figure 9-2. Distribution of utterances in Bloom's Taxonomy while adding new functionality. Participants had not previously carried out an inspection.

The Application level contained the second highest utterance count on average. This is the level where developers make the actual changes to the code, implementing the new functionality. Two participants actually had more utterances at the Comprehension level than at the Application level. This indicates that they were still attempting to grasp the code as they were explaining it, via thinking aloud.

Figure 9-7 shows the utterance break down from the participants who had conducted an Ad hoc inspection prior to adding new functionality. These participants also had the largest percentage of utterances, on average, at the Knowledge level, followed by the Application level. This is similar to the participants who had not executed a code inspection previously, but the Knowledge category was lower and the Application category was higher. The prior exposure to the code assisted the developers to move more rapidly into the changes reflected in operating at the Application level.

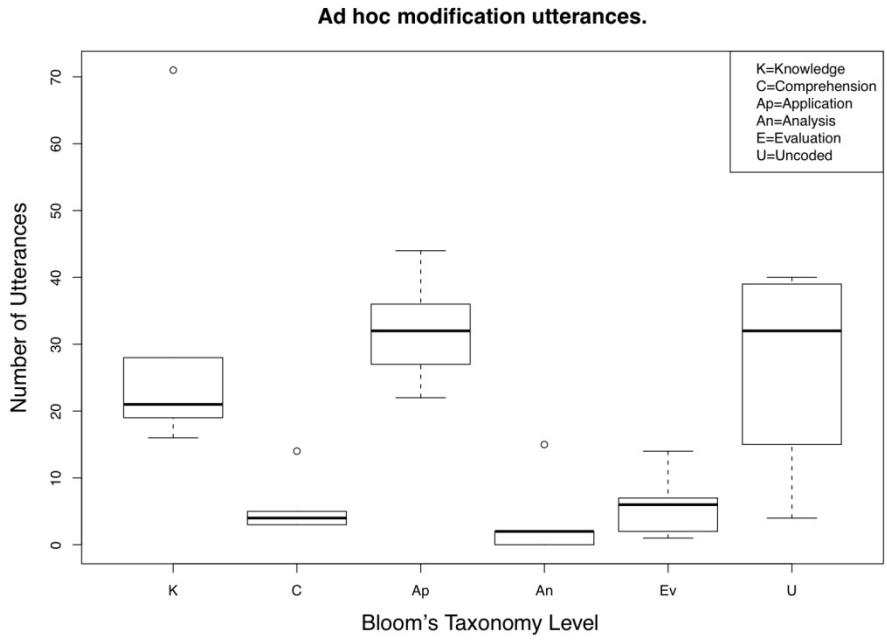


Figure 9-3. Distribution of utterances in Bloom' Taxonomy while adding new functionality. Participants had previously carried out an Ad hoc inspection.

Figure 9-8 and Figure 9-9 display the utterance breakdown from the participants who had previously completed the ADR and CBR inspections respectively. From these two graphs, it can be seen that the percentage of Knowledge utterances drops again, compared with those who did not inspect the code previously and those who performed the Ad hoc inspection. In a similar fashion, the Application category is

higher in both groups.

These results indicate that participants who carried out a structured code inspection prior to adding new functionality tended to operate at the higher cognitive levels.

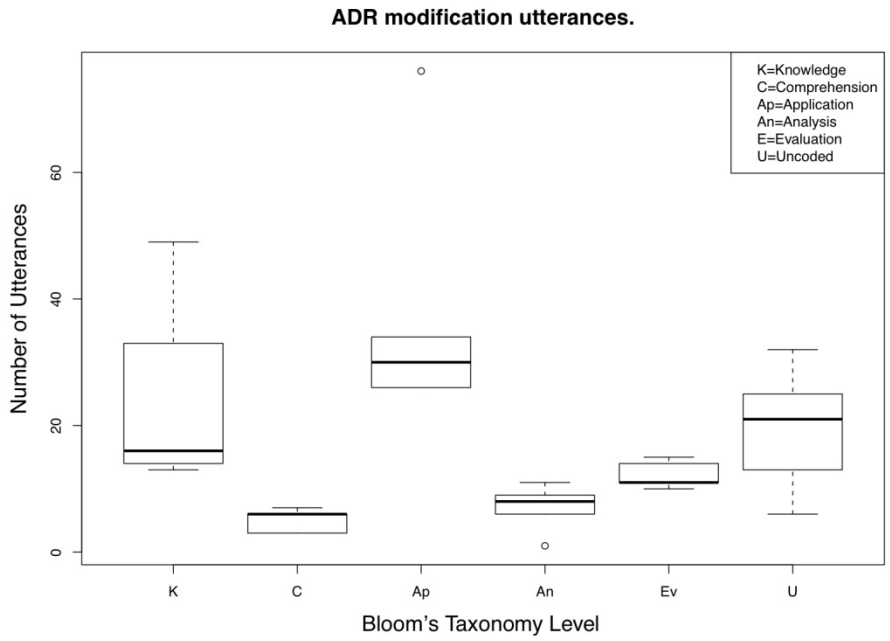


Figure 9-4. Distribution of utterances in Bloom's Taxonomy while adding new functionality. Participants had previously carried out an ADR inspection.

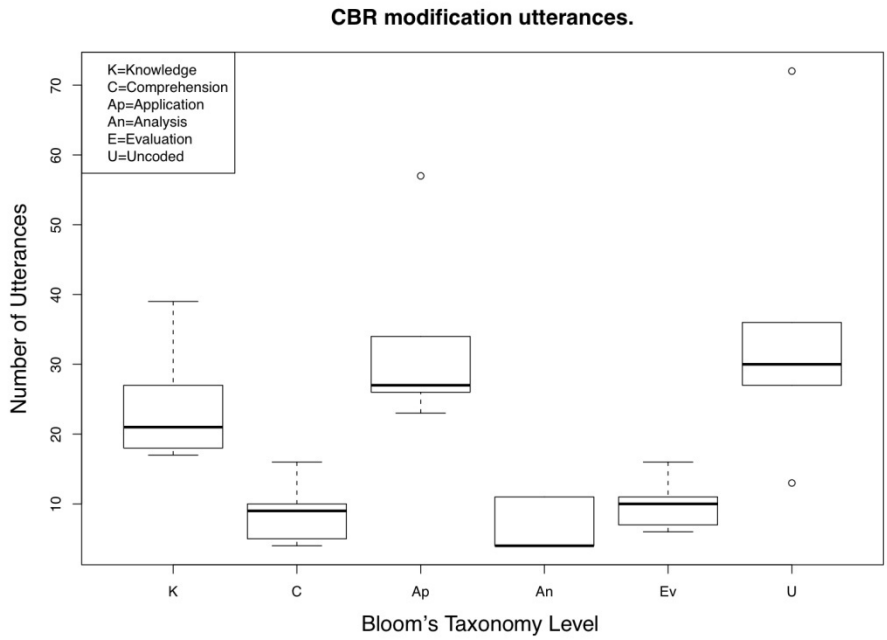


Figure 9-5. Distribution of utterances in Bloom's Taxonomy while adding new functionality. Participants had previously carried out a CBR inspection.

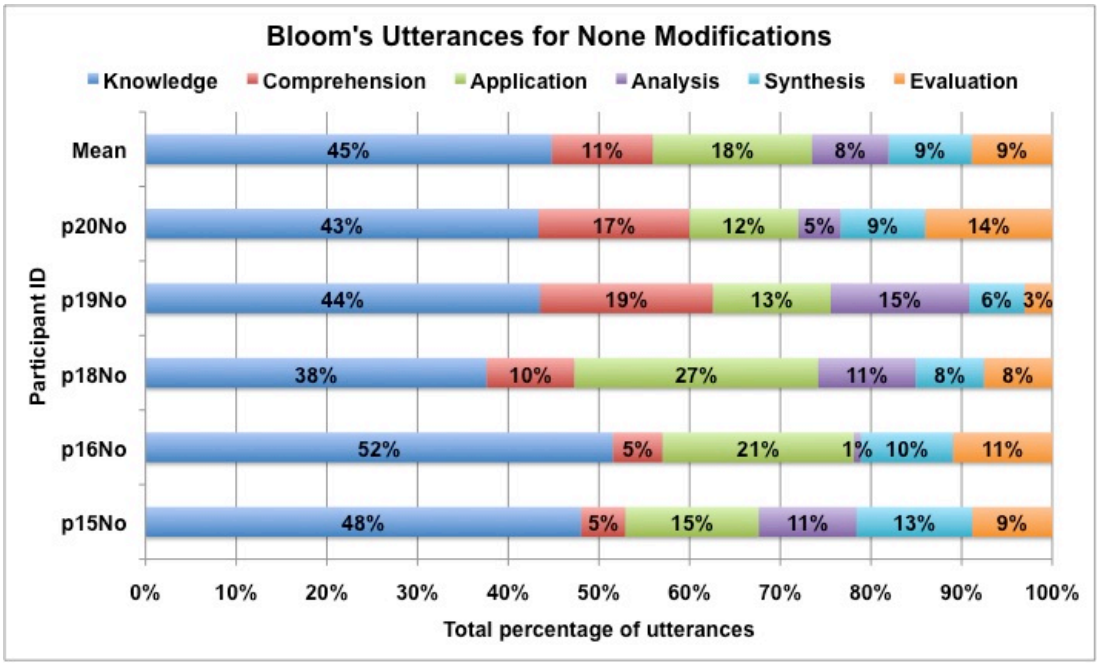


Figure 9-6. Breakdown of each participant's utterances while adding new functionality (previously performed no inspection).

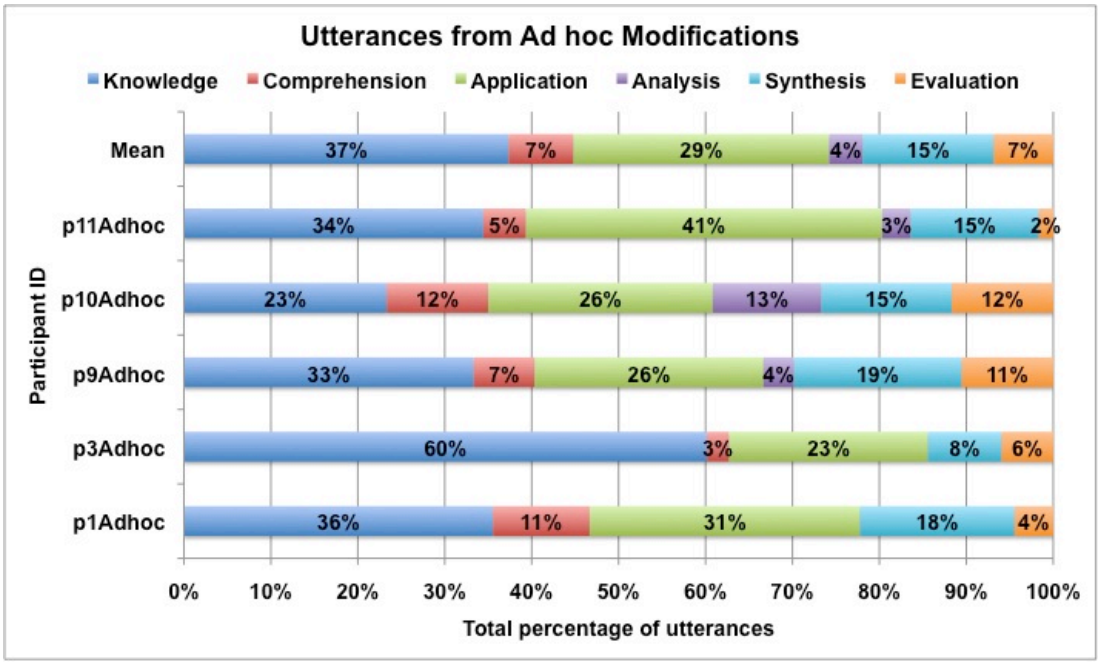


Figure 9-7. Breakdown of each participant's utterances while adding new functionality (previously performed an Ad hoc inspection).

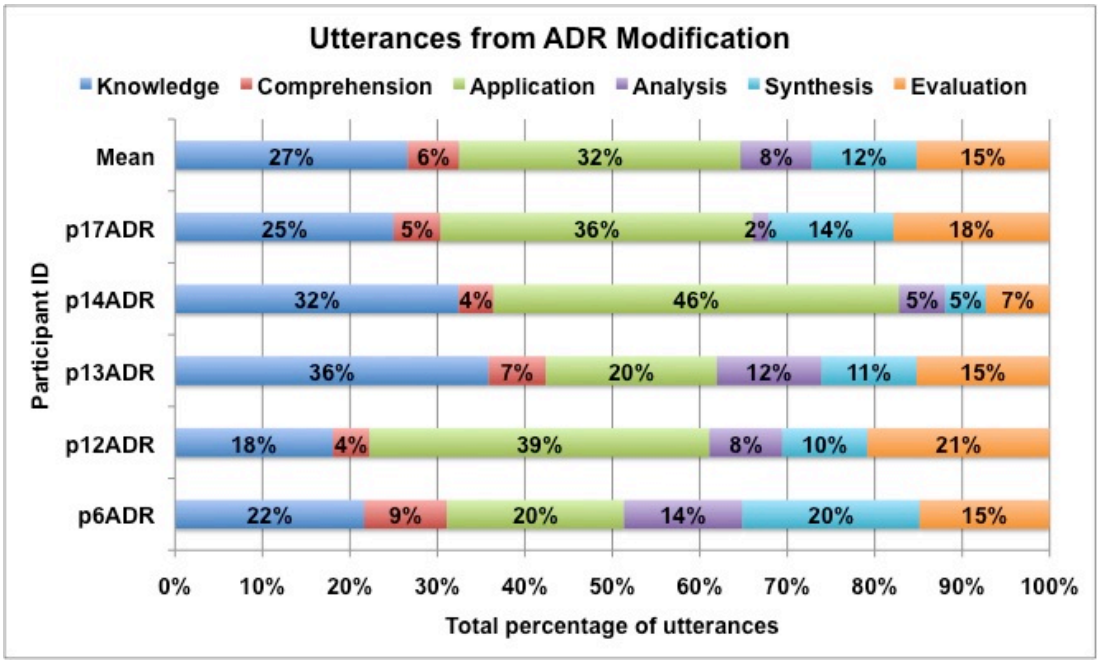


Figure 9-8. Breakdown of each participant's utterances while adding new functionality (previously performed an ADR inspection).

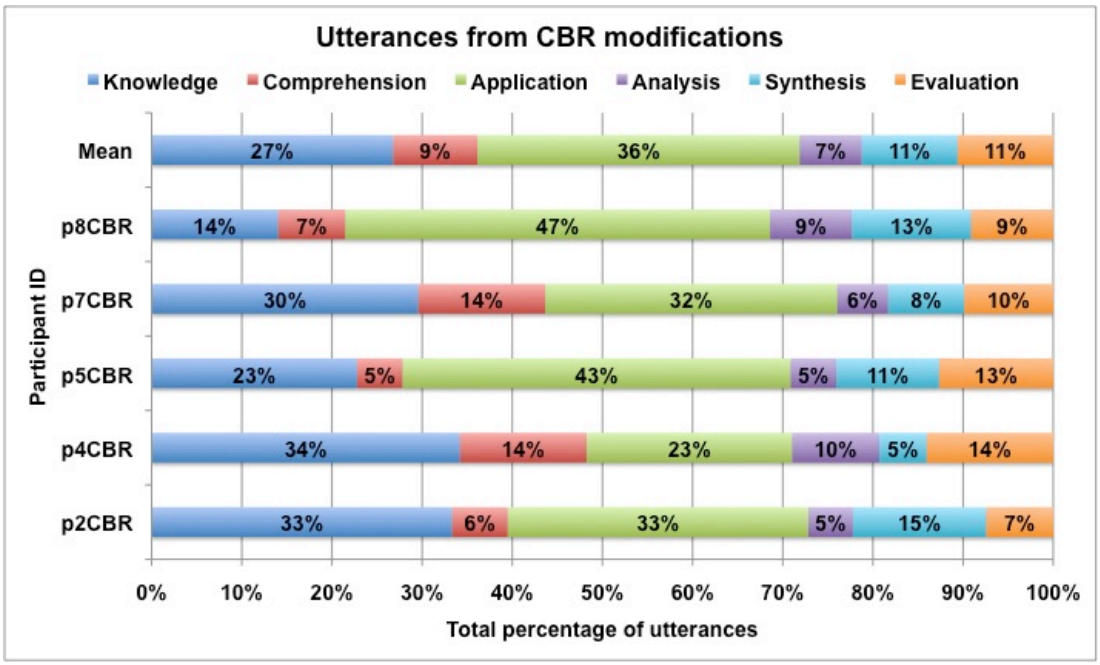


Figure 9-9. Breakdown of each participant's utterances while adding new functionality (previously performed a CBR inspection).

The order of utterances of four different participants is graphed in Figure 9-10 to Figure 9-13. The X-axis shows the order of the utterances and the Y-axis represents the utterance's cognitive level according to Bloom's Taxonomy, with "0" equating to the Uncoded category.

Participant 15's modification utterances are shown in Figure 9-10. Almost 50% of utterances for this participant were at the Knowledge level. Prior to starting the task, this participant was completely unfamiliar with the code. Therefore, in the 30 minutes given for the task, he needed to both familiarise himself with the code, and add the appropriate functionality.

Figure 9-10 can be considered as a time-line. The participant commences the task at the Knowledge level and remains at this level for some time. Once the code has been sufficiently understood, the participant attempts to add the new functionality. At this

point, s/he begins to operate at higher levels of thinking. This pattern was similarly repeated with all participants who did not perform a code inspection prior to undertaking the task of adding the new functionality.

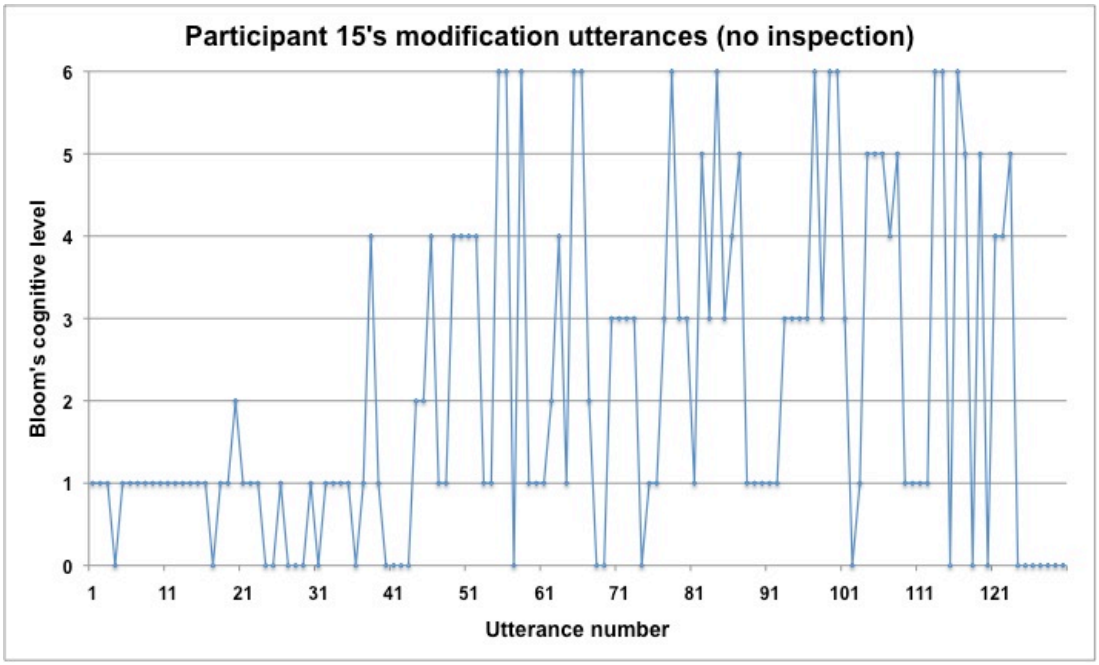


Figure 9-10. Participant 15's order of utterance categorised using Bloom's Taxonomy.

Figure 9-11 shows the order of participant 5's utterances. The graph shows participant 5 starting the task with a small number of utterances in the low Knowledge cognitive level. This was quite brief and s/he moved to functioning at the higher cognitive levels: Application and Evaluation. The utterances remained at those levels for most of the remaining time. This pattern was also reflected in the other participants who performed the CBR inspection prior to the changes.

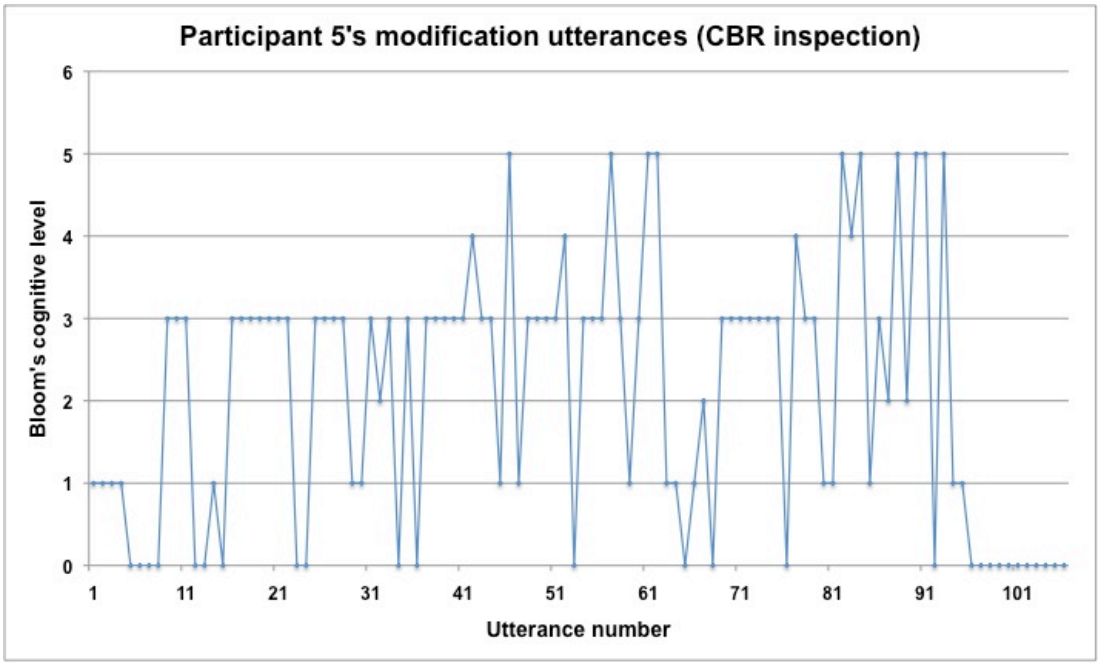


Figure 9-11. Participant 5's order of utterances categorised using Bloom's Taxonomy.

Participant 11's utterances in order of occurrence are displayed in Figure 9-12. Participant 11 starts with two Knowledge utterances, moves up to the Evaluation level and then moves steadily in the lower Knowledge level and then moves up to the higher levels of Application, Evaluation and Synthesis. Participant 11 functions reasonably consistently at the Application level. This pattern was similarly reflected by the other participants who also carried out the Ad hoc inspection prior to adding functionality to the code.

Participant 17 carried out an ADR inspection prior to adding functionality to the code base and their utterance pattern is displayed in Figure 9-13. The graph is similar to participant 5's and participant 11's, shown in Figure 9-11 and Figure 9-12, where there are a few low level utterances to begin with and then they move up to the higher cognitive levels to carry out the remaining task.

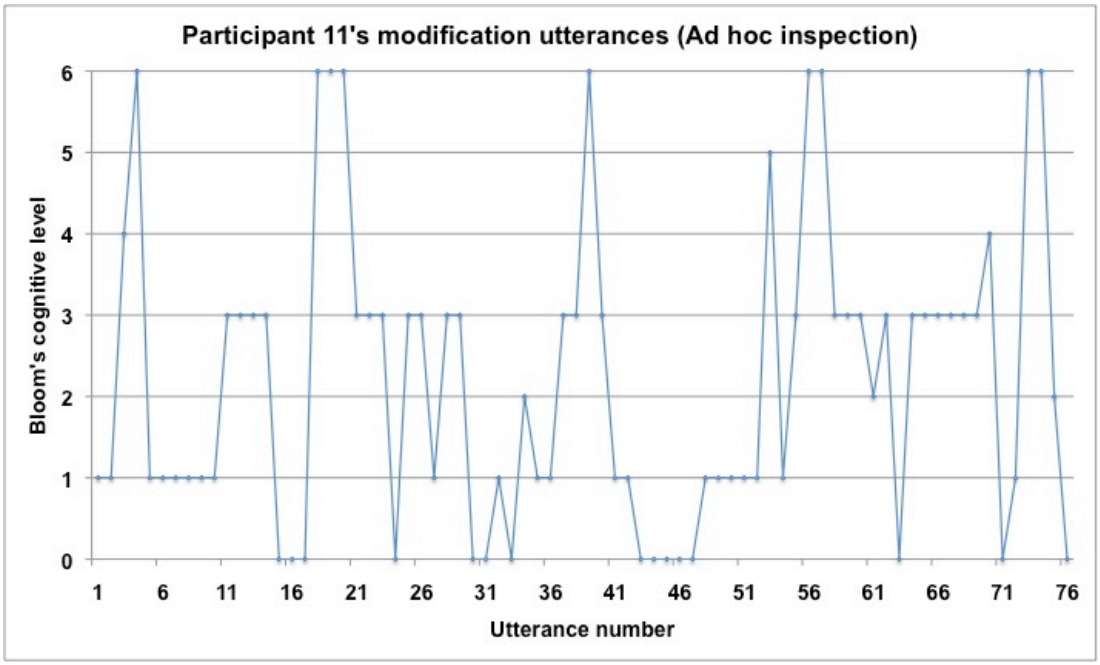


Figure 9-12. Participant 11's order of utterances using Bloom's Taxonomy.

Figure 9-14 displays participant 1's utterances from the Ad hoc code inspection they performed prior to adding the new functionality. This figure and Figure 9-10 look quite similar in the way that they commence. In both they start at the Knowledge level and remain there for some time. Approximately half way through the task, participant 15 begins to move into higher levels in order to carry out the code changes. Participant 1, as seen in Figure 9-14, has the vast majority of their utterances at the lower cognitive levels, Knowledge and Comprehension.

These two situations are similar in that the Ad hoc inspection technique is without structure and guidance and most student inspectors tend to operate at the lower cognitive levels. When attempting to make changes to an unfamiliar code base with no guiding methodology, the student developer tends to function for quite some time at the lower cognitive levels. This could be related to the inspection technique or to their limited experience. The cognitive patterns, the order of their utterances,

displayed in both of these situations are similar. During the inspection, participant 1 is working to understand the code while participant 15 is working to add functionality to the code. When 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.

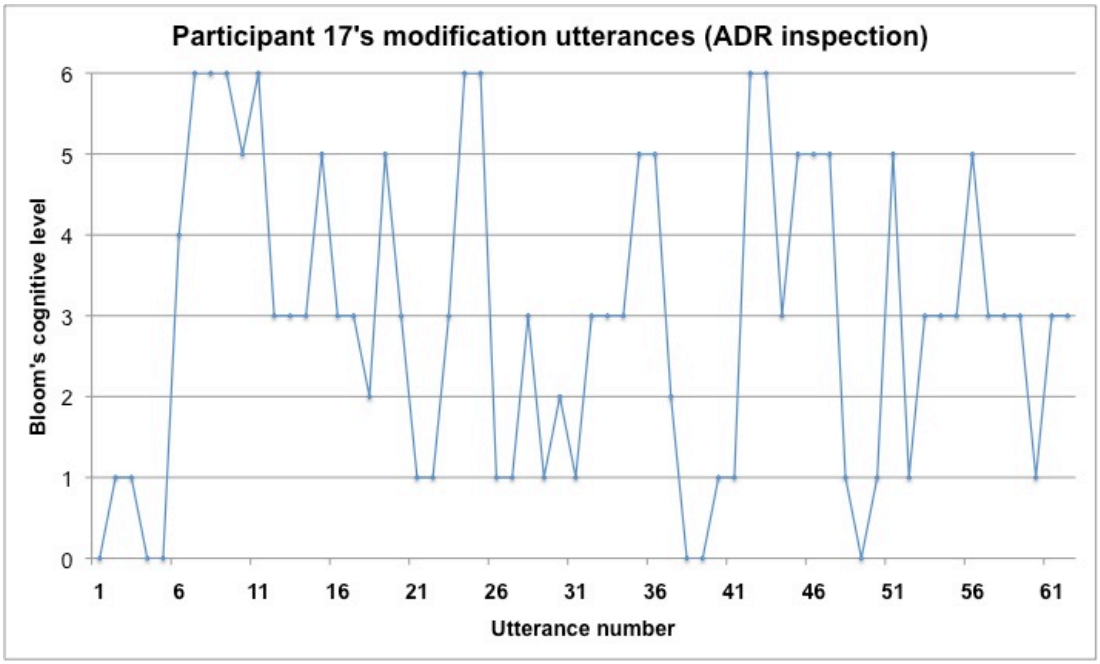


Figure 9-13. Participant 17's order of utterances categorised using Bloom's Taxonomy.

9.4 Discussion

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

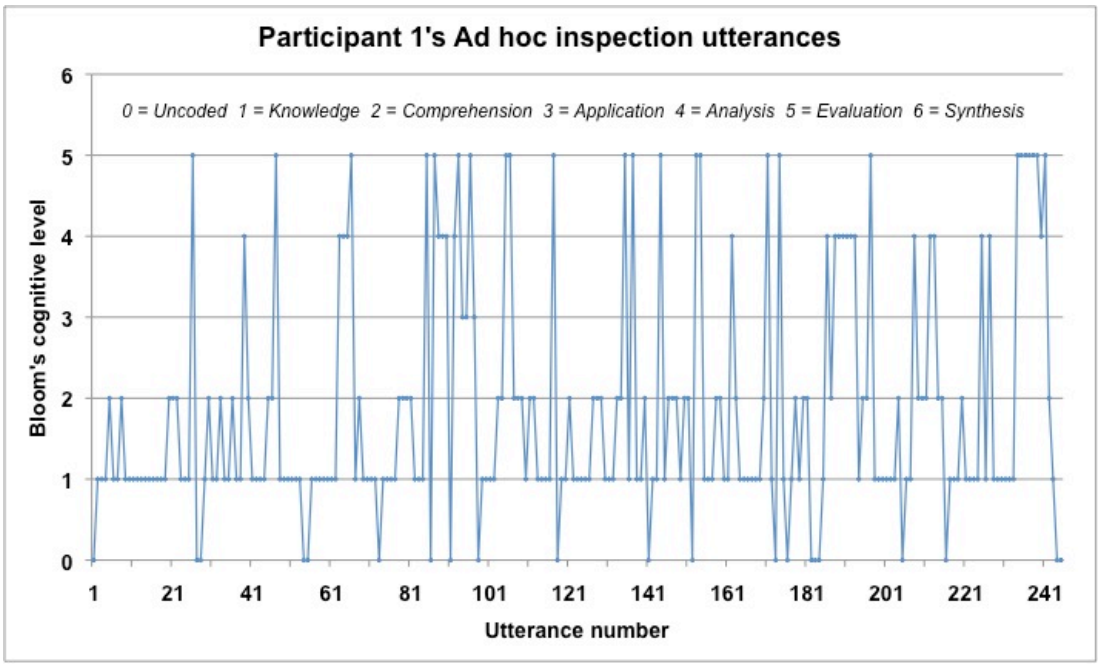


Figure 9-14. Participant 1's ordered utterances from the Ad hoc inspection.

Figure 9-10 shows that 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 9-14. Notably, these two participants operated at similar cognitive levels, although they performed very different tasks: participant 1 performed an Ad hoc code inspection and participant 15 attempted to add new functionality to the code base.

Participant 1 used the unstructured Ad hoc inspection technique, relying on the inspector's own experience to execute the inspection successfully. The cognitive levels experienced when carrying out an Ad hoc inspection are similar to those experienced when adding new functionality to unfamiliar code.

With the Ad hoc inspection technique, the inspector is given no direction regarding what to do or how. In the case of adding new functionality, the code is unknown and the developer must build an understanding in order to successfully add the

functionality. However, when participant 1 moved into the modification task, s/he operated at the higher cognitive levels of Application and Synthesis. This started very early, unlike participant 15 when s/he performed the same task.

From the results shown in the previous chapter, it appears that the participants who implemented the CBR inspection technique operated at higher cognitive levels than did those who did not implement that technique. When the CBR inspectors then moved to add new functionality to the code, they continued to operate at the higher cognitive levels: Application, Synthesis and Evaluation. This generally occurred more consistently and with more utterances. 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.

9.5 Conclusion

The experimental results indicate that novice/student developers, who perform a code inspection prior to implementing changes to the code, tend to operate at higher cognitive levels from very early in the change process. Those who had not performed the inspection tended to operate at lower cognitive levels for longer periods of time.

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.

The results highlight the important role that 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 first goes 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.

10.0 Chapter Ten

Cognitive Process Models of Software Inspection Techniques

10.1 Introduction

In this chapter, the software inspection techniques investigated in this thesis will be mapped to the cognitive process model the technique facilitates the inspector to use. It will also map the inspection technique to the level of Bloom's Taxonomy that is best supported by the software inspection technique. These two mappings will be based on: 1) the descriptions of how to apply the inspection technique, 2) the specific tasks that the inspection technique requires the inspector to perform, and 3) the results from the earlier experiments.

The mappings will help to address the research questions regarding the cognitive process model that the different inspection techniques facilitate and the level of Bloom's Taxonomy at which they require inspectors to operate.

With this knowledge, software development teams may be able to decide which techniques are best suited in their situation to assist developers to operate at higher cognitive levels.

10.2 Ad hoc

The Ad hoc inspection technique is a common, informal inspection technique that is widely used (Laitenberger, & DeBaud 2000). There are no guidelines with this technique and an inspector is expected to find as many defects as possible using his/her personal experience as a guide.

The Ad hoc inspection technique appears to facilitate the inspector to implement the "As needed" cognitive process model. This is a logical conclusion as the inspector searches the code in whatever manner s/he is accustomed to in an attempt to detect defects. Hence, the code is searched as needed in order to detect the defects.

However, this may not be the way in which the inspector goes about searching for defects. Although the Ad hoc technique is considered a common technique, with no given structure, the developer may go about searching for defects in a manner that reflects a different cognitive process model and not the "As needed" way. Because the technique relies on the inspector to use his/her experience, the inspector may actually implement the inspection in a manner that more accurately resembles one of the other cognitive process models. For this reason, the Ad hoc technique is difficult to categorise because it depends upon the inspector's personal experience and how s/he goes about searching for defects. Therefore, it can be accurately categorised only on a case-by-case basis.

With no guidance provided, the experienced inspector is expected to carry out the technique in some kind of systematic manner. However, the novice inspector would be expected to function in a manner that more closely resembles the "As needed"

model. That is, they begin at an arbitrary location and move through the code in whatever way they believe to be appropriate.

For example: The data collected from the experiments showed one novice inspector who was assigned the Ad hoc technique systematically worked through the code, asking questions about the code at different stages. This inspector worked part-time in industry and mentioned on several occasions that this was how they inspected code at work.

Another novice inspector assigned the Ad hoc inspection technique, who had not worked in industry, found this technique very difficult and inspected the code with little direction or purpose.

These two examples indicate that experience has a large influence on how one goes about reading and inspecting code.

Therefore, from the literature review and the results from the experiments reported within, the cognitive process model supported by the Ad hoc inspection technique is difficult to classify, but may reflect the "As needed" model.

The Bloom's Taxonomy level facilitated by the technique is difficult to classify except on a case-by-case basis. The experiments in this thesis, however, found that the Ad hoc inspection technique required the inspectors to mostly operate at the Knowledge cognitive level.

10.3 Checklist-Based Reading technique

Figure 10-1 is a sample question from the checklist used in this thesis. In order to answer the question with a “yes” or “no”, the inspector needs to know what the starting condition is, examine the changes that the code can implement upon the data and check whether, following any changes to the code, the execution sequence is correct, thereby allowing this part of the program to function correctly. The inspector must understand the intended meaning of the code and verify if the intended meaning corresponds to what actually occurs.

19	Is the correct sequence of code executed for any condition outcome?			

Figure 10-1. An example of a question from the checklist used in this thesis.

Hence, a small amount of code is chunked together and examined. This process continues method by method until the entire class has been inspected. Once the inspection of the class has been completed, the inspector will have a list of defects and/or can state whether or not the class functions correctly.

This inspection technique reflects the bottom-up cognitive process model whereby the developer groups small sections of code together in order to understand the code. Implementing the Checklist-Based Reading technique, the questions require the developer to group small amounts of code together to validate it against the checklist question.

Figure 10-2 depicts this process, starting from the bottom with a small chunk of code, then progressing upwards grouping chunks together and answering the

checklist questions.

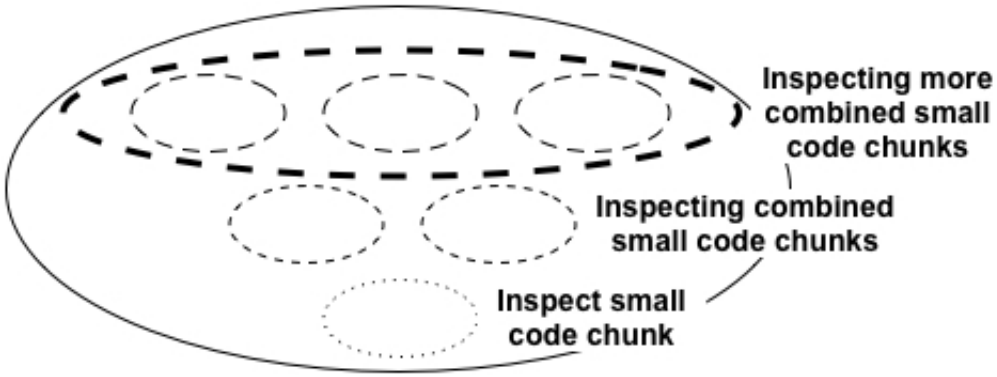


Figure 10-2. Bottom-up cognitive model process as implemented using the CBR inspection technique.

Therefore, both the literature and the results of our experiments indicate that the Checklist-Based Reading technique supports a bottom-up cognitive process model.

Examining the technique in the context of Bloom's Taxonomy, the Checklist-Based Reading technique facilitates inspectors to function at the Evaluation cognitive level. The results from our experiment indicated that this was the case; the technique facilitated the inspectors to operate at the Evaluation level.

10.4 The Abstraction-Driven Reading technique

The Abstraction-Driven Reading technique, designed for object-oriented systems' code, requires the inspector to write a small abstract that describes each method within the inspected class. This process is repeated until the entire class and its functionality has been described. The inspector identifies whether or not the code is fulfilling the purpose it was intended for while examining the code in order to write each abstract.

The Abstraction-Driven Reading process starts at the low-level of code. The inspector reads the code, chunking it together, usually according to specific methods, and writes an abstract describing what the method does. This process is repeated for each method until all methods within the class have an abstract and then an abstract for the overall class is written. The inspector examines a small code segment within a method, writing a description of what that code does. Another code segment is then examined and a small abstract describing that segment is written. This continues until the entire method has been described. Then, an abstract is written that describes the entire method. The abstracts that were written about the code segments making up to the overall description are then discarded. An example follows.

```
1  private static void humanPlayerTurn( Player human, Player computer )
2  {
3    boolean continueAttack = false;
4    int validAttack        = -1;
5    int attackLocation []  = new int[2];
6
7    while( !continueAttack )
8    {
9       attackLocation = playerAttackLocation();
10      validAttack    = human.getBombedLocation( attackLocation[0],
11      attackLocation[1]);
12
13      if( validAttack == 0 )
14      {
15          computer.attacked(attackLocation[0], attackLocation[1]);
16          human.setTurns();
17          continueAttack = true;
18      }
19      else
20      {
21          System.out.println( "Location attacked, try another!" );
22      }
23   }
24 }
```

Figure 10-3. A code segment.

Figure 10-3 contains a code segment from a larger Java class file. The following describes the steps taken using the Abstract-Driven Reading technique to create the

overall abstract that will describe this method.

The **humanPlayerTurn(…)** method receives two **Player** objects, **human** and **computer** as parameters. The **humanPlayerTurn(…)** method then creates and initialises several local variables.

The local variable **continueAttack**, originally initialised to false, is used as the condition in the **while** loop. Now, as long as **continueAttack** is false, the method remains in the **while** loop.

The variable **attackLocation** is assigned the returning value from the **playerAttackLocation()** call. The **playerAttackLocation()** method gets the row and column coordinates from the human **Player** which they intend to attack. The coordinates chosen by the player are then checked for validity inside the **Player** object via the call to the **getBombedLocation()** method.

Up to this point in the inspection, the abstract shows that it is the human player's turn and s/he has entered coordinates to attack on the computer player's board.

If the entered coordinates are valid, then the computer **Player** is attacked through a call to the **attacked()** method located in **Player**, and the computer **Player** is attacked and the result of this is displayed. The turn counter is incremented indicating that it is the computer **Player**'s turn and the variable, **continueAttack**, is set to true allowing the **while** loop to exit.

The final abstract becomes similar to this: the `humanPlayerTurn(…)` method is responsible for the human player's turn. It gets the attack coordinates that the human player wants to attack, validating them and attacking the computer player's board at those coordinates, allowing the success or failure of the turn to be displayed and sets the program to the computer player's turn.

The example Abstraction-Driven Reading process described in the preceding paragraphs shows that this reading technique has the inspector start at the code level, working upwards towards a more general description. First, the description is about a couple of lines of code, then it is about several sections of code and it continues until the entire method has been described. This process is repeated for each method within the class until an overall class description can be written. The Abstract-Driven Reading technique facilitates the inspector to operate using the bottom-up cognitive process model.

The Abstraction-Driven Reading technique requires the inspector to read the code and write an abstract describing the code's functionality. The inspector describes the code and its intricacies when summarising each code fragment, and these are combined into an overall summary description of the class.

For inspectors to be able to do this they need to know what it is that all the code used within the method and class does. As seen in the `humanPlayerTurn(…)` example above, at several points in time the inspector needed to move from the method being examined to other methods within the same class. They were also required to move from the method they were examining, and also to other classes due to methods

being called upon different object types.

Line 15 `computer.attacked(…)` required the inspector to move to the `Player` class. The method `attacked(…)` within the `Player` class made a call to the `receiveAttack(…)` method within the `Board` class. Hence to understand the code within the `humanPlayerTurn(…)` method the inspector also needed to understand code segments from two other classes within the overall system.

Therefore, examining the Abstract-Driven Reading technique in the context of Bloom's Taxonomy, it appears to facilitate two of the Taxonomy's cognitive levels. One level is the Comprehension level. This level represents the ability to describe what is communicated in a different manner. The Abstract-Driven Reading technique requires the inspector to read the code and then communicate it in a different manner, in the form of a natural language abstract. The second level facilitated is the Analysis level. By tracing the code within a method to different methods within the same class and also through to other classes within the system, the inspector examines communication between the method and the wider system. The communication is broken into different parts in order to write an accurate account within the abstract. Hence, this reflects the Analysis level within Bloom's Taxonomy.

The results from the experiment actually showed that the ADR technique facilitated inspectors to operate mostly at the Knowledge level and then the Comprehension level. The Analysis level was quite low. There are two possible reasons for this. First, the technique requires the inspector to take quite some time to carry out the

inspection and write the abstracts. Second, the inspectors were completely unfamiliar with the technique until they had carried out the training exercise. It may be that this technique has a steeper learning curve than the experiment permitted.

Hence, even though the ADR technique appears to require inspectors to operate at the Comprehension and Analysis levels, this was not confirmed by the experiment. This inspection technique will need further research.

10.5The Use-Case Reading technique

The Use-Case Reading technique, designed for use in Object Oriented systems, requires the inspector to trace a use case scenario through a sequence diagram. At each point where the code under inspection is called, the inspector directly refers to that code, inspecting it to ensure that it fulfils its intended purpose.

The Usage-Based Reading technique is almost exactly the same as the Use-Case Reading technique. The principal difference between these two techniques is that the Usage-Based Reading technique prioritises the use cases. The use case with the highest priority is the one that is perceived to be of most importance to the user. That is, if this use case fails, it will cause the most inconvenience to the user.

A use case "describes the system's behavior (sic) under various conditions as the system responds to a request from one of the stakeholders" (Cockburn 2000, p. 1). Depending upon the request made and the conditions within the system at the time of request, the system will respond with varying behaviour sequences or different scenarios. The use case captures these scenarios and behaviours (Cockburn 2000).

The use case is generally a textual format describing what is occurring in the system. The textual description contains enough information for a developer to understand what is going on, and yet it contains as little technical language as possible in order for non-technical people to be able to understand what is being described.

The use case describes a program section, what causes it to begin, what occurs during the process and what state the program is in when it completes. The inspector takes this information and then verifies and validates the program against this, deciding whether or not it actually performs the tasks correctly.

Due to the similarities between the Use-Case and Usage-Based Reading techniques, the example included in this section is valid for both techniques, assuming that this is a prioritised use case within the system.

Figure 10-4 displays the Use Case diagram for an audio player system. The “Selects next track” scenario is highlighted, as this is the scenario that will be examined in this example. Table 10-1 shows the “Selects next track” Use Case scenario. The use case scenario describes the steps that the system will undertake in order to fulfil this selected scenario. The scenario’s Sequence Diagram is shown in Figure 10-5.

The inspector begins by reading through the use case scenario. This explains what the user is doing and how the system is meant to respond to the user. The inspector starts at the use case diagram and notes that it is the “User selects next track” scenario that has been chosen. Next, the inspector reads the use case. The scenario shows that a random order track is currently playing. The steps taken by the system

to fulfil the user's choice are listed in the Main success scenario.

The inspector steps through the scenario, sequence diagram and code to determine if the code is correctly executing the scenario and sequence diagram.

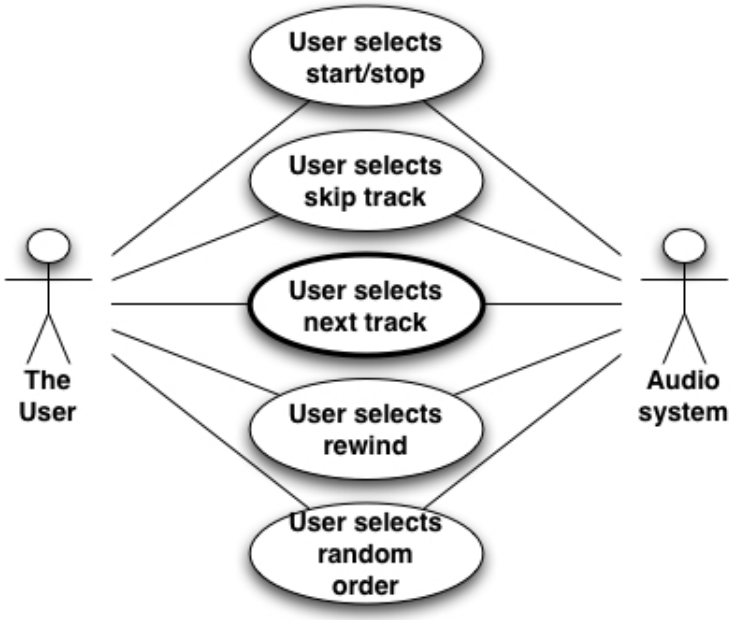


Figure 10-4. Use Case diagram of audio player system, user selects next track scenario selected.

Table 10-1. Use Case scenario, select next random track.

Use Case Scenario: Select next track
Primary actor: The user.
Precondition: A random order track is playing.
Main success scenario:
1. The user selects next track.
2. The system checks if there is another track in the play list.
3. The system randomly selects a track from those still in the play list.
4. The system plays the next random track.

In this example, the inspector starts by checking the `UserInterface`, examining the code related to calling the next track. Then, the inspector moves to the `Player` class

and then, according to the sequence diagram, into the **`RandomProgramme`** class. While examining the code, the inspector notes where the code is not correctly carrying out the task. That is, the inspector makes notes of any defects encountered.

The Use-Case and Usage-Based Reading techniques begin with the inspector looking at a certain part of the system's functionality. In this example, the audio player was playing music in a random order and the next track option was selected. Functionality is described, and then the inspector moves from this high level description down to a lower level, code level inspection. The inspector then moves back to the high level description and then back down to the lower level code inspection.

The inspection starts out with a top-down cognitive process model. The top-down cognitive process model is where an overall hypothesis describing the system is created. This hypothesis is refined by smaller hypotheses being created describing sections of a system. This breakdown continues until the system is completely understood. The Use-Case and Usage-Based Reading techniques facilitate this top-down cognitive process model. The inspector is presented with a high level abstraction describing a scenario of the system's usage. From this, s/he traces the scenario through the sequence diagram. When the sequence diagram refers to the code being inspected, s/he then moves to that code, inspecting it to ensure it is correctly carrying out the task that is supposed to be performing.

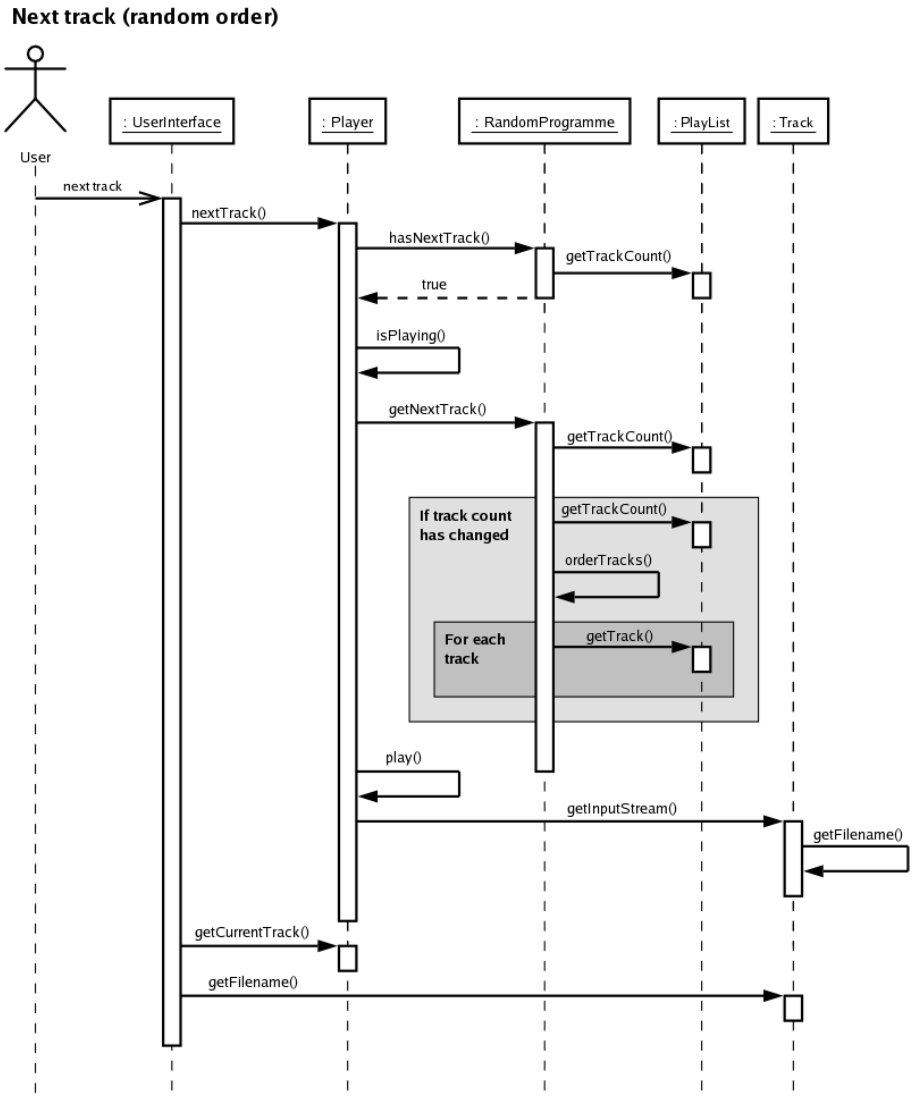


Figure 10-5. The Sequence Diagram for the Select next track while playing in random order, scenario.

Code inspections will eventually lead to a bottom-up cognitive process. This is because the principal goal of the code inspection is to ensure that the code carries out the task for which it was intended. That is, it is necessary to determine whether or not the low level code fulfils the purpose for which it was designed.

However, the Use-Case and Usage Based-Reading techniques shown in the example begin with facilitating a top-down cognitive process model. As already noted,

applying these two techniques means that not all code in a class is inspected. That is, only the code involved in the scenario is being inspected. The "As needed" cognitive process model is one whereby the developer examines and learns only the code in the immediate vicinity related to the task they need to perform. These two reading techniques are similar to that. Only the code involved in a scenario is inspected. Hence, these two reading techniques also facilitate the "As needed" cognitive process model.

Carrying out a UCR or UBR inspection requires the inspector to understand a use case scenario and the sequence diagram that represents it. From the scenario and the sequence diagram, the inspector moves to inspect the code. In the process, the inspector takes the use case and the sequence diagram and then translates them into the code representation. The inspector takes those original descriptions and translates them into another representation. Within Bloom's Taxonomy, this is representative of the Comprehension cognitive level.

Also, the sequence diagram shows interactions between the different objects within the scenario that come from different parts of the system. To be able to verify the correctness of the code, the inspector needs to understand the relationships within the system represented in the use case scenario and on the sequence diagram. This requires an understanding of how the different objects interact with each other and their overall roles within the system. To be able to do this, reflects the Analysis level of Bloom's Taxonomy.

Hence, the Use-Case and Usage Based-Reading techniques facilitate both the

Comprehension and Analysis levels described in Bloom's Taxonomy.

The experiment reported in this thesis showed that the UBR inspection technique facilitated the Comprehension level, in that on average, this level contained the highest number of utterances. The Analysis level was also facilitated but on average there were more Knowledge level utterances than there were Analysis level utterances.

Table 10-2. Summary of different software inspection techniques and the cognitive process models they facilitate.

Technique	Cognitive Model	Bloom's Taxonomy Level		Justification
		Expected	**Reported**	
Ad hoc	NA	NA	NA	No methodology is provided, inspectors use their own experience. As no support is provided, it is totally up to the inspector
CBR	Bottom-up	Evaluation	Evaluation	Start with the code and check that the code works correctly and then move up to higher levels.
ADR	Bottom-up	Comprehension, Analysis	Knowledge, Comprehension	A very structured way to read the code is provided. Start with code and then map this to natural language that should match the specification.
UCR	Top-down & As needed	Comprehension Analysis	Comprehension Knowledge, Analysis	Start with use cases that describe system functionality and work from that to the actual code.
UBR	Top-down & As needed	Comprehension Analysis	Comprehension Knowledge, Analysis	Start with use cases describing system functionality and work down to see if it is correct. Originally this was proposed for use with design documents.

10.6Summary

Table 10-2 presents a summary of the five different inspection techniques examined within this thesis. It presents a summary of the cognitive process models they

support and the Bloom's Taxonomy cognitive levels facilitated, both the expected level and the level that was recoded within this thesis.

10.7 Conclusions

In this chapter, five software inspection techniques have been described. The descriptions related to the cognitive process models that each technique requires the inspector to implement during an inspection. The description also examined each technique and which cognitive level, as described in Bloom's Taxonomy, it most closely facilitated the inspector to operate at during a software inspection.

This knowledge and understanding can now be used to assist software development teams in providing an understanding about the particular inspection techniques that facilitate specific cognitive process models. It also gives these teams information about which techniques facilitate the inspector to operate at higher cognitive levels.

In an educational setting, such as at a university, it is now known what levels within Bloom's Taxonomy of Educational Objectives each reading technique facilitates. When creating tasks within curriculums, it is possible to match tasks and desired outcomes with the reading technique that best facilitates the appropriate cognitive process model and Bloom's Taxonomy level.

In the next chapter, recommendations for reading guidelines and inclusion of software inspection techniques into degree and training programs as a reading technology to improve software developer comprehension will be made.

Bibliography

Cockburn, A., 2000, *Writing Effective Use Cases,* Addison-Wesley.

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

11.0 Chapter Eleven

A Code Reading Strategy

11.1 Introduction

Ultimately, the executing code is the most up-to-date software artefact found within a software system. Therefore, a software developer's ability to read code is an essential skill that needs to developed and cultivated throughout a career in software development. Guidelines for developing this skill need to be developed and expanded in the software development discipline.

Gabriel and Goldman (2000) and Spinellis (2003) point out that the software development industry is one of the few, perhaps the only creative fields, that does not actively promote reading the work of others. In this context, that means reading the code of others. However, there is a problem with this. In some instances, the only way to read the code would be through decompiling it. In many cases, this constitutes a breach of a license agreement. In some countries, this would breach many laws and the developer could face a large fine and/or a possible jail sentence. However, the increased creation and use of Open Source software is changing the extent to which code written by others is available for reading. Through Open Source, developers now have the opportunity to read the code bases of many high quality products such as Firefox, Thunderbird, Linux kernel, the KDE and GNOME

desktops, to name just a few. Open Source code also provides developers with the opportunity to read source code that is not well written or constructed.

11.2 Reading technologies

Within this thesis different software inspection reading technologies have been examined, related to the possible significant use of them in the non-traditional area of software developer comprehension.

The results from the experiments reported within this thesis indicated several things:

- The inspection technique itself had no significant impact on the number of defects detected by both student and industry professionals;
- Industry professionals found that the more structure in an inspection technique, the more restricted they felt in attempting to thoroughly search the code for defects;
- Student developers found the more structure an inspection technique had, the more confident they felt to carry out an inspection;
- Student developers felt that having performed a prior code inspection, they were better equipped to make changes to the code as well as add new functionality to it;
- Implementing a UBR inspection, mapping a sequence diagram to the underlying code, with no previous experience with the system, does not facilitate inspectors to function at higher levels of thinking: and
- Checklist-Based Reading facilitated inspectors to operate at higher cognitive levels, according to Bloom's Taxonomy, when carrying out a software inspection and adding functionality.

In the software development life cycle, maintenance often fails because the persons assigned to maintain the system has not fully understood the software. They make changes to the system without understanding that the area they have made changes to is tightly coupled with other areas within the system. Hence, the change made in one location can affect the system in several other locations.

Reading code is a time consuming and laborious task; nevertheless, it is a vital task. With 70% of a software system's budget being spent on maintenance and system evolution (Bennet 1990; Pfleeger Lawrence, & Atlee 2010; Swanson, & Beath 1989), and between 50% and 90% of this time spent by developers reading and attempting to understand the code (1996; 1994; 1996; 1993; 2000), any significant changes that can be made to assist developers to operate at higher cognitive levels and improve their understanding, have the potential to significantly reduce costs and improve software quality. There are no shortcuts and no silver bullets for this process, but there are approaches that can assist to increase one's efficacy in this area. Software inspection reading techniques is one approach that can assist here.

During the learning process, the learner must pass through each of the cognitive levels described in Bloom's Taxonomy. The learner begins at the lower levels and progresses up through the different levels towards the higher ones. This is the same for a software developer in the learning process. They move through each cognitive level, from Knowledge to Synthesis, increasing their understanding of the system as they go. The application of software inspection reading techniques as reported in this thesis aims to increase the efficacy of the developer's learning, at each stage of this

process.

11.3 Guidelines

A goal in training and education is to lead people from the point at which they are currently at to a place of mastery, equipped with the necessary skills and knowledge to make this progression. The following guidelines are aimed at achieving this goal, to move software developers from the place they are currently at along the journey towards becoming masters of their profession, software development. These guidelines arise from the results of the research described in this thesis.

"Brookes Law" states that adding new software engineers to a software project team that is already behind will put it even further behind. One reason for this is the increased communication overheads placed upon the senior developers (Brooks 1995). Also, there is a ramp-up time that developers must pass through in order to become productive team members. This is regardless of whether or not the developer is experienced or is a novice.

Implementing structured reading strategies with new team members may assist in reducing this ramp-up time as well as reducing the burden placed on senior team members. New team members build and cultivate their understanding and comprehension of the system, in part, by implementing the reading strategies.

11.3.1 The novice developer

As is the case in most fields, when one is new to something, one's knowledge and

understanding is limited and therefore more structured assistance is required. The application of systematic software reading through performing software inspections is one way in which developers new to a project can gain a focussed understanding of the system they are working on to increase their knowledge and understanding of the system.

From the work reported in this thesis, it appears that novice developers required structure to support their cognitive development. Novice developers stated that they preferred structured methods when inspecting code for defects. The results indicated that the novice developers who carried out inspections using more structured methods, operated at the higher cognitive levels than those novice developers who had used the less structured methods.

Novice developers will pass through all the cognitive levels described within Bloom's Taxonomy as they learn. The studies' results have shown that developers start at the lower cognitive levels and progress upwards through the taxonomy. This progression through the different stages is not a new concept. Education and other research disciplines have been reporting this for some time.

Asking a novice developer to read, understand, and then make changes to a software artefact may be a daunting task. However, giving the novice developer a software artefact and asking him/her to inspect it for defects using a defined inspection technique, provides the novice with three things:

1. Clear instructions as to what to do;
2. Instructions as to how to do it; and

3. A metric enabling him/her to ascertain how well s/he is carrying out the task.

It must be noted that failure to detect defects does not necessarily mean that the code is defect free. It means the inspector did not find any. Upon completing the inspection, the novice has an increased understanding of the code. From our experimental results, the novice is now more confident to carry out maintenance and evolution tasks. Also, with their increased system understanding gained from the inspection, the problem of maintenance failure due to developers' lack of system understanding may be reduced or avoided.

11.3.2 The non-novice developer

The research reported within this thesis indicated that the more experienced a developer is, the less direct support structure they require for reading and examining software systems. Their experience has given them the necessary skills to carry out the task.

An experienced developer starting a new project may be handed a software artefact such as code, and told to read it, understand it, and then make specified changes to it. This is less likely to be as daunting for them as it would be for the novice developer. This is because, through their experience, developers have developed their own methodologies, processes and techniques to carry out such a task.

The experienced developer can still benefit from structured reading techniques in this situation. The software inspection reading techniques have been shown to increase developers' cognitive levels at which they operate while carrying out

different software engineering tasks. They assist the developer to move systematically through the different cognitive levels, acquiring a working understanding of the program.

11.4 Software inspections as a reading technology

Students starting university, technical training or participants in software development training courses come from different backgrounds and experience levels. Some have experience with software development, while others have none or very little prior knowledge. It is most important to introduce developers to the skills required for writing and reading code. Currently, many courses teach people to write software. Writing software is important. Reading software is also important. At some point in a software developer's career, s/he will be required to read software that was written by someone else.

The ability to read code written by someone else is an important skill to be developed. Code reading is a skill that should be explicitly taught in degree and training programs. The omission of such a skill in curricula means that it is then up to the student to acquire it "on the side" or "on the job".

Implementing structured reading methodologies throughout degree courses/units is one way to increase students' ability to integrate into an existing software development team. With these skills, their ability to operate at the higher cognitive levels will assist them to become integrated, productive team members as they make the transition to professional practice.

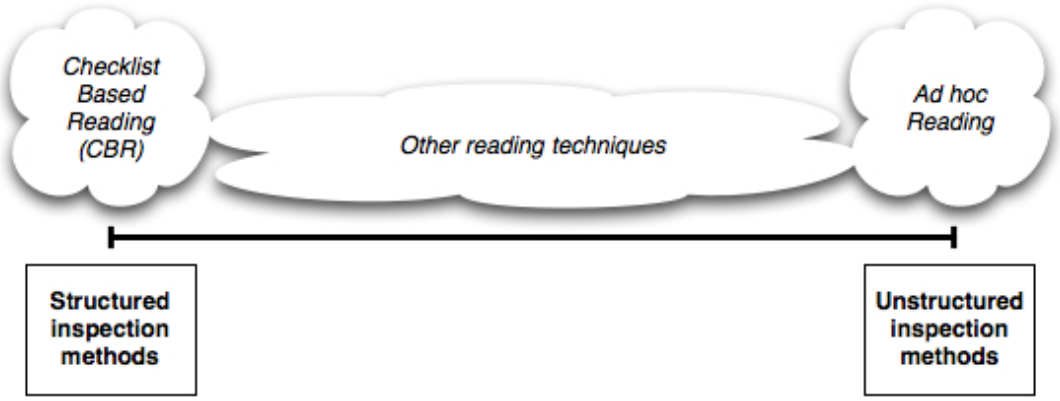


Figure 11-1. Reading technique continuum.

Figure 11-1 shows a software inspection reading technique continuum. It commences with the most structured reading technique on the left, through to the most unstructured technique on the right. In the context of this thesis, the Checklist-Based Reading technique is considered the most structured for several reasons. This technique provides inspectors with the questions that they need to answer along with the instructions. Hence, the process is defined and structured.

Figure 11-1 represents a reading technique continuum, with the starting point being Checklist-Based Reading, the most structured technique, through to the Ad hoc Reading technique at the other extreme of the continuum as an unstructured reading technique. This is because the Ad hoc technique provides the inspector with only the artefact under inspection and the instruction to find as many defects as possible. The inspection process details are left to the inspector, who must rely on prior experience to carry out the task. Hence, this is considered the most unstructured inspection technique.

On the continuum between Checklist-Based Reading and Ad hoc reading lie the other inspection techniques. The techniques explored for the purposes of this thesis

are at different points along the continuum. The placing depends on at least:

1. The amount of structure provided by the technique; and
2. To what degree is it reliant on the inspector's level of personal experience.

11.4.1 The learning curve

When a student enrols in a university course or some kind of software developer training program, the course usually caters for those with varying levels of prior knowledge and experience. During the course of his/her studies, the student's knowledge and understanding is expected to increase. At the start of the learning curve, a large amount of foundational learning occurs. This is especially so for those starting with little or no background knowledge or experience. The introduction of structured reading techniques at this stage into courses/units will benefit the students in several ways by giving them foundational code reading skills that they can build on as they move along the learning curve.

Over time, the amount of structure provided by the reading technique is reduced. This is based on the assumption that the new student is not very familiar with the art of reading software code. Therefore, a high degree of structure is imposed by an outside source: the software reading technique. Over time, as the student becomes more familiar with the reading of software code, the amount of structure provided by the reading technique is reduced. Each stage builds upon, and continues to implement and take advantage of, the previous reading techniques. This allows the student to increasingly use his/her own experience to tackle the task at hand, in terms of what and how things need to be done.

Table 11-1 proposes where the software inspection reading techniques, studied in this thesis, could be implemented within degree programs such as Software Engineering, Computer Science, Information Technology and Information Systems. The table is based on the explanations already presented, starting with the highly structured techniques at the beginning of the degree program when the students are still acquiring their foundational knowledge. As the degree program progresses, along with the students' ability to function more effectively within their chosen courses, the amount of structure within the reading techniques being taught is reduced. The previous techniques are not removed from the courses. As shown in Table 11-1, they remain, but the less structured reading techniques that focus the student on the behaviour of the system, take higher precedence.

Table 11-1. Proposed introduction and placement of reading techniques within degree programs.

Degree Year	Reading Techniques	Un/structured
First	CBR, ADR.	Highly structured
Second	UCR, CBR, ADR.	Semi structured
Third	UBR, UCR, CBR, ADR.	Semi structured
Fourth	Ad hoc, UBR, UCR, CBR, ADR.	Unstructured

11.4.2 The learning curve joining an existing project

A professional developer starting on an unfamiliar project, will have a ramp-up time; that is, the time it takes for him/her to become a productive member of the team. In this situation, structured reading may assist the developer to increase the efficacy of this ramp-up time. The majority of developers join teams that are already working on existing products. With maintenance consuming up to 70% (Bennet 1990; Pfleeger Lawrence, & Atlee 2010; Swanson, & Beath 1989) of a software system's lifetime, and 50% to 90% of this time estimated to be consumed by developers attempting to

understand the system in order to perform the needed maintenance, increasing developer efficacy in this area is vital (1996; 1994; 1996; 1993; 2000).

11.5 Software inspections as a comprehension technology

The proposed introduction and placement of reading techniques is displayed in Table 11-1. This section will discuss the integration of the techniques in Table 11-1 within courses and how each technique builds upon the foundation established by the previous technique.

11.5.1 CBR as a program comprehension technology

This research has demonstrated that the CBR technique facilitates inspectors to operate at the Evaluation level within Bloom's Taxonomy. The application of this technique, even while developers are gathering low-level knowledge, can be beneficial. The questions on the checklist require the inspector to function at the Evaluating level even while low level cognition is taking place.

The repeated application of this technology while the developer is learning, assists in building the foundational skills of critical analysis of the code they are reading. The repetitiveness of applying this technique may become a habit, a step on the journey from novice to master developer.

The CBR technique, because of its highly structured nature, is introduced as the first reading technique.

11.5.2 ADR as a program comprehension technology

The ADR technique is introduced as the developer's experience and understanding increases, building on the CBR foundation. The CBR technique required inspectors to answer "yes" or "no" to questions. The ADR technology requires inspectors to write natural language abstracts describing the functionality of the code. The abstracts are written commencing with the methods with the least dependencies within the classes with the least dependencies.

The ADR technique in principal facilitates the Comprehension level of Bloom's Taxonomy. In applying the ADR technique, the developer is acquiring and/or consolidating the skill of reading and understanding the interactions of the code within a system.

The application of the ADR technique is combined with the application of the CBR technique, already established by the developer. Hence, the ability to read code and write abstracts describing the functionality is combined with the question asking habit established by the CBR technique. The continued application of the ADR technique builds a habit in the developer of describing what the code does and how it does it.

The repetitive application of these reading techniques is establishing foundational skills of reading, describing, and evaluating code. These are essential skills for both inspecting code to evaluate if it is fit for the intended purpose, as well as to understand how code interacts with the wider system.

Establishing this kind of system understanding will assist developers to maintain and evolve systems on which they are working

These two reading techniques, CBR and ADR, establish developer habits whereby they acquire their understanding using the bottom-up cognitive process model.

The ADR technique is a structured technique but does rely on the developer being able to read and understand code. Consequently, it is placed as the second reading technique to be introduced into courses, following on from the CBR technique.

11.5.3 UBR and UCR as program comprehension technologies

The introduction of UBR and UCR techniques builds on the CBR and ADR techniques' established foundations. In this thesis, it has been shown that the application of the UBR and UCR techniques differs only in regards to the use case prioritisation.

In applying UBR and UCR, the developer commences with a high level description, a use case scenario, of a system interaction. These techniques facilitate the developer to move from a high-level description to a lower level, the code executing the use case scenario.

While applying these two techniques, the developer calls on those skills already acquired from the CBR and ADR techniques. Reading through the scenario and sequence diagram, the developer's habit of asking questions about what the scenario is doing and why it is doing it, in conjunction with the creation of descriptive

abstracts, is helping the developer to understand how specific interactions are executed within the system. These combined skills provide the developer with tools that s/he uses in order to increase their system understanding and make them adept in identifying if the code is fulfilling its intended purpose. This increased system understanding will also assist developers when maintaining and evolving these systems.

The UBR and UCR techniques enable the developer to acquire and enhance his/her understanding of the system by using a top-down cognitive process model.

The UBR and UCR techniques are semi-structured reading techniques. When applied, these techniques omit code from an inspection that is not executed within the use case scenario. Therefore, it is important that they be applied in conjunction with other inspection techniques. This requires the inspector to use his or her own experience to see what code has been omitted from the inspection. As these two techniques rely on a developer's experience, it is recommended that they be introduced later in a course, after the CBR and ADR techniques.

11.5.4 Ad hoc reading as a program comprehension technology

The Ad hoc technique is the least structured of the reading techniques but was favoured by the experienced developers during the experiment presented in this thesis. This reading technique relies solely on the developer's experience to read and understand the code.

The developer commenced his or her code reading with the highly structured and

directive CBR technique that tends to focus on the software's structure. Following this, the ADR technique was introduced, requiring the developer to describe in his or her own words what the code was doing.

The UBR and UCR techniques direct the developer to examine and read the code from a behavioural, interacting perspective. The scenario describes an interaction within the system and the developer examines the code from the point of view of the scenario executing.

Due to the experience and expertise gained by the developer, having applied the different reading techniques over a period of time, the developer is able to read code in the manner that s/he has developed through the systematic introduction of the different reading techniques. This then becomes the developer's personal Ad hoc reading technique to extend their understanding of a software system.

The Ad hoc technique is the least structured technique, relying on the developer's own experience and consequently, within the structure of a training course, this is the last reading technique to be introduced.

In a systematic manner, through the repeated application of these reading techniques, the novice developer is led towards mastery of the craft.

11.6Conclusion

This chapter has recommended the introduction and implementation of software inspection reading techniques in degree programs (as well as training courses) to

build the essential skill of reading code that every software developer will need to have at some point in his/her career as a software developer.

As the software systems being built continue to simplify the user experience, the complexity for developers is increasing. Therefore, it is important that effective methods be developed to assist developers, both novice and experienced, who are joining teams, to understand the code bases on which they are working as quickly and as efficiently as possible.

Software inspection reading techniques may be implemented as effective training methodologies for experienced developers new to a project. By assisting the developers to operate at higher cognitive levels, it may reduce lead time and allow them to apply to the current project the experience they have accumulated during their time in the industry.

Bibliography

Bennet, K.H., 1990, An introduction to software maintenance, *Information and Software Technology*, 12(4), pp. 257-64.

Brooks, F.P., 1995, *The Mythical Man-Month, anniversary edition,* Addison--Wesley, San-Francisco, USA.

Canfora, G., Mancini, L. & Tortorella, M., 1996, Proc. Fourth Workshop on Program Comprehension, *A workbench for program comprehension during software maintenance.* pp. 30-9.

Dunsmore, A., Roper, M. & Wood, M., 2000, The Role of Comprehension in Software Inspection, *The Journal of Systems and Software*, 52(2--3), pp. 121-9.

Gabriel, R.P. & Goldman, R., 2000, Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications, *Mob software: The erotic life of code.* Mineapolis, U.S.A.

Hartzman, C.S. & Austin, C.F., 1993, Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research CASCON '93, *Maintenance productivity: observations based on an experience in a large system environment.* IBM Press, pp. 138-70.

De Lucia, A., Fasolino, A.R. & Munro, M., 1996, Proceedings of Fourth Workshop on Program Comprehension, 1996, *Understanding function behaviors through program slicing.* pp. 9-18.

Pfleeger Lawrence, S. & Atlee, J.M., 2010, *Software Engineering Theory and Practice,* Fourth (International) ed. Pearson, Upper Saddle River, N.J.

Rajlich, V., 1994, Proceedings of Summer School on Engineering of Existing Software, *Program Reading and Comprehension.* Bari, Italy, pp. 161-78.

Spinellis, D., 2003, *Code Reading: The Open Source Perspective,* Addison-Wesley

Professional.

Swanson, E.B. & Beath, C.M., 1989, *Maintaining information systems in organizations*.

12.0 Chapter Twelve

Recapitulation and Future Work

12.1 Introduction

Software has become ubiquitous, and the world we currently live in is highly dependent upon computer systems, controlled by software, for its every day functioning. A computer failure in today's world can bring about something as simple as the twirling beach ball on the Mac OS or the blue screen of death on a Windows OS, through to the failure of autopilot systems in a state-of-the-art aircraft, to name just a few. Computer systems do fail due to hardware malfunctions, but often the impact of something like this is reduced through redundant hardware. Computer systems more often fail because of an error in the code that is running the system and that error was introduced through a human error in its programming. The errors are often caused because the developers sometimes fail to understand the system or the impact that changes to the code base will have upon the system.

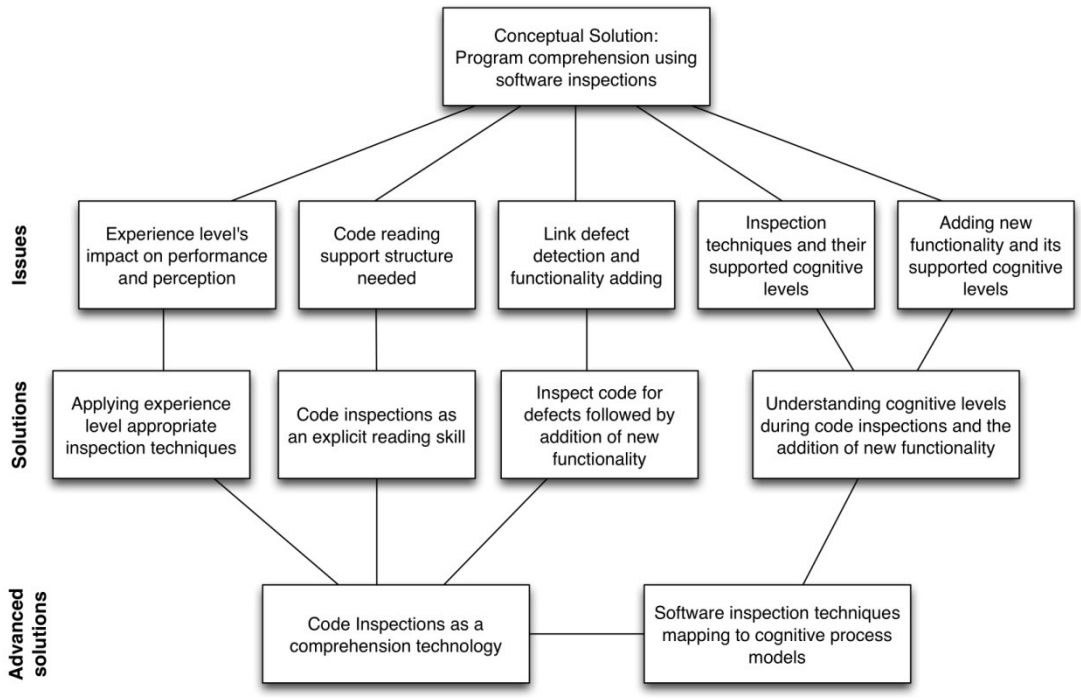
Effectively increasing developers' cognitive ability to understand and comprehend the program as we have described that to be can have lasting positive consequences, less failure within software systems that impact upon the society in which we live.

This chapter provides a recapitulation of the work that has been carried out in

producing this thesis. The research questions and issues identified and posed within Chapter Three will be summarised in relation to how they were addressed through this thesis. Contributions that this thesis makes to the wider body of knowledge will be discussed followed by recommendations for future research.

12.2 Recapitulation

In recapitulating the work of this thesis, Figure 4.3 is included here as this laid the foundation of the issues and solutions that were addressed. Chapters Five to Nine addressed the issues listed along the Issues row providing solutions to each. Chapters Ten and Eleven brought together all of the work, linking it to form the cohesive Advance Solutions identified on the bottom row of the Figure.



Chapter One highlighted the significant role that software plays in today's society. Almost all aspects of the industrialised world have been influenced by sophisticated software systems. A software failure in one of these systems can have a major impact in today's world. With this important role that software plays, Chapter One examined several software development process models that have evolved, and where this research fits in the software development life cycle.

The cognitive process models implemented by developers as they attempt to learn and understand the system they are working on was introduced along with Bloom's Taxonomy and how it will be used within this research.

In Chapter Two, a literature review was conducted to examine the previous work within fields related to the work undertaken for this thesis. Software inspections and the impact these have had on software production were examined. This was followed by a review and evaluation of several different software inspection techniques reported in the literature.

Program comprehension and the five cognitive process models implemented by software developers as they learn the system they will work on was reviewed, with it being noted that between 50% and 90% of maintenance time is consumed by developers attempting to understand the system. The bottleneck for further advancements within software development has been set to lady with developers, the human ability to carry out the tasks needed for these advancements (Kothari 2008).

An analysis of the use of protocol analysis or think-aloud data was conducted. This

analysis demonstrated that much of what is known about program comprehension has stemmed from this data collection method. Of late, this data collection method has been combined with Bloom's Taxonomy in order to analyse the data.

Research questions arising from the literature review became the basis for Chapter Three where the research problems addressed in this thesis were defined.

In Chapter Three, the problems that were addressed within this thesis were formally defined. Chapter Three defined and clarified the key concepts that were used within this thesis. Program comprehension was defined as a key challenge within the software development life cycle with the need to improve it being isolated as an important issue for software engineering research to tackle. Therefore, the research issues that were addressed within this thesis were defined in order for them to be clarified so that they may be addressed. Addressing these issues then formed the body of this thesis in Chapters Five through Eleven.

An explanation and description of possible research methodologies that could be applied to address the research issues was then presented. This is concluded with an explanation regarding the research approach/philosophy/direction that was adopted for the purposes of this thesis.

Chapter Four provided the overview of the solution to the research issues that were defined in Chapter Three. A conceptual solution framework was presented that indicated how the issues were related and how the solutions provided, tied together to contribute a possible solution to the problem of software developer

comprehension.

Program comprehension was shown that it could mean different things at different times during the software development life cycle. Program comprehension was described in the context of what it means within this thesis. As it can mean different things at different points along the software development life cycle, it was described in the context of the Analysis, Evaluation and Synthesis levels of Bloom's Taxonomy.

Chapter Five addressed the issue of the impact that software developers' experience level has on both the performance and the perception of a software inspection. Three different software inspection techniques were compared when implemented by novelists or student developers and by industry professionals. From the three techniques tested, no one technique out-performed another when implemented by novice or professional developers. However, there was a significant difference when comparing the novice developer with the industry professional. Beyond the significance of the number of defects detected, industry professionals found the CBR technique to be restrictive, not allowing them to use their experience gained.

The student developers, however, liked the CBR technique as the structure provided aided them to carry out the inspection. The professional developers preferred less structure in the inspection technique while the novice developers preferred more structure.

This finding becomes significant in that, although the different inspection techniques

were not appreciably different with respect to their ability to detect defects for inspectors of similar experience, experienced developers found the highly structured CBR inspection technique too restrictive while the novice developers found it very supportive.

In the application of software inspection techniques to assist to improve developer cognition, developer experience levels need to be considered. For the experienced developer, a very structured technique can restrict the developer from applying their experience to improve their comprehension of a system. However, if the developer lacks experience or is a novice, it is important that the inspection technique provide an adequate structure to support them to carry out the task.

Chapter Six examined the cognitive levels expressed by inspectors as they mapped a sequence diagram to the underlying code that executes the described scenario. The hypothesis examined in this experiment, was that the nature of this kind of task correlated well to facilitating the inspector to function at the Analysis level within Bloom's Taxonomy. However, this was not the case as the task facilitated developers to operate at the Knowledge and Comprehension levels.

Participants also failed to accurately map the sequence diagram due to underlying, executing code. With the participants who took part in this experiment, the task was not achieved.

The false positives made by the participants indicated that having developers perform this task, on completely unfamiliar code, did not facilitate operating at the

Analysis level. This kind of task demonstrated that some familiarity with the code prior to performing the activity might have been beneficial to the developers. This prior familiarisation may have allowed them to function at higher cognitive levels. It is important that tasks to familiarise developers with the code base allow offer them to pass through the lower cognitive levels before attempting to have them function at higher cognitive levels.

In Chapter Seven, the results from an experiment are reported. The experiment examined whether or not a correlation exists between the number of defects an inspector detects during an inspection and the number of successful changes, through adding new functionality, that the inspector makes to the inspected code base.

It also reported on the different ways developers went about adding the new functionality to the code depending on whether or not they had performed the code inspection previous to adding the new functionality. Participants also answered questions regarding their perceptions of the way that the software inspection assisted them.

The results showed there to be a weak correlation between the number of defects detected by the inspector and the number of successful changes they make to the code. The weak correlation may have also been related to the small sample size within the experiment.

The results regarding how the participants went about carrying out the code changes indicated, within the given situation and sample size, showed that those who had

performed the inspection implemented them significantly closer to the ideal solutions provided. Intuitively, this makes sense as the code base is now familiar to the developer because they have carried out an inspection prior to making the changes. Already having an understanding of the code through prior inspection means the inspectors have understood the code and have verified whether or not it implements what it was designed for. Although this may seem intuitive, a formal process can be established and then monitored within the software development life cycle.

Developers all expressed that: they found performing the inspection assisted them to make the required changes, it would now be easier to add new classes to the system and that it enabled them to better understand the system. Hence, it can be seen that carrying out an inspection assisted developers in changing the code, navigating through and around the code and gave them more confidence to work with the code.

Chapter Eight then addressed the issue of the different cognitive levels at which inspectors operated while carrying out a software inspection implementing a specified software inspection technique.

The Ad hoc, CBR and ADR inspection techniques were used within the experiment. The results indicated that the Ad hoc inspection technique, the least structured of the techniques, facilitated the inspectors to mostly function at the Knowledge cognitive level. The Ad hoc inspection technique relies on the inspector's personal experience to successfully carry out the inspection. In this experiment, the participants were novice inspectors, hence the experience level was low. Without support from the

inspection technique, the inspectors were left to work things out for themselves. This supported the results reported in Chapter Five in which the novice developers expressed that they found the unstructured inspection techniques aren't helpful because of the lack of structure.

The ADR technique facilitated inspectors to operate, on average, mostly at the Knowledge cognitive level. The next level most supported was the Comprehension cognitive level.

In the ADR technique, the focus is for inspectors to create a natural language abstraction for each method. One is also created for the entire class. The nature of this method reflects the Comprehension cognitive level. However, the results from this experiment, in this situation, did not demonstrate this to be the cognitive level at which inspectors mostly operated. This may have resulted from the time permitted for the task, 30 minutes. Inspectors may not have had long enough to move fully into the abstract writing task, hence the Comprehension level was not the level at which the developers functioned most often. The inspectors may still have been gathering and building their foundational knowledge.

The CBR technique facilitated operating for the vast majority of the time at the Evaluation level. At this level, the inspector is evaluating the code for correctness and omissions. The CBR technique provided the inspectors with questions that need to be answered. The nature of the questions required them to examine the code in respect to the question. Hence, the method is assisting the developer to perceive the intended meaning and verify whether it is doing what it was intended to do. As

shown in Chapter Four, the CBR technique facilitates this to occur.

It was also the CBR technique that novice developers in Chapter Five, mentioned they preferred to use because of the provided support.

Chapter Nine addressed the issue of the cognitive levels developers expressed while adding new functionality to code previously inspected. Participants from the experiment described in the previous chapter, upon completing the software inspection, were presented with a request to add new functionality to the code. A new group of participants took part in this section of the experiment. These new participants had not conducted the inspection previously, and also had not seen the code prior to participating.

Experimental results indicated that participants who had carried out the inspection prior to adding new functionality tended to operate at higher cognitive levels beginning earlier in the task. Adding new functionality to a code base requires the developer to operate at several cognitive levels.

In making the changes, the developer needs to operate at the Application level. It is at this level that the developer writes the code. The Synthesis level is required as this is where the functionality is created. Evaluation is also needed to decide if the new functionality works as intended. The experiment results show this to occur. The vast majority of utterances by developers who did not perform code inspections were at the Knowledge level. This was in order to familiarise themselves with the code prior to adding the new functionality. Those who had inspected the code had already

gained this experience.

The cognitive levels of the Ad hoc inspectors were similar to the levels of the participants who had not performed prior code inspection. In both cases, they worked for some time at lower levels. The Ad hoc inspectors had no support structure or guidance, but had to rely on their personal experience. The participants who had not carried out the inspection were also reliant on their own experience to lead them through the changes they needed to make.

Without the experience, or a structure to support them, the developers function at the lower cognitive levels for some time when compared with participants who had conducted the prior inspection. Ad hoc inspectors also functioned for longer at the lower Knowledge cognitive level. Although there was a familiarity with the code, it did not appear that this was as strong as for those who had carried out a structured inspection.

The tables Table 12-1 to Table 12-5 display tabular summaries of each experiment conducted during the course of this research. The results section highlighting that which was discovered through conducting the experiment, building upon the results of the previous experiment(s).

Table 12-1. Summary of Chapter 5's experiment results.

Experiment Details	Summary
Aims	To identify the impact inspector experience has upon software inspections. To compare differences between the Checklist-Based Reading (CBR), Usage-Based Reading (UBR) and Use Case Reading (UCR) inspection techniques in identifying defects and the impact inspector experience has on their efficacy.
Participants	Industry professionals and undergraduate students.
Artefacts	Java software system created by Dunsmore et al. (2002)
Results	Kruskal-Wallis test to determine any statistical difference between the inspection techniques for: # of defects detected by technique: Asymptotic Sig. = 0.4 # of delocal defects detected by technique: Asymptotic Sig. = 0.7 # of false positives generated by technique: Asymptotic Sig. = 0.1 Mann-Whitney test was to determine any statistical difference between the participant groupings for: # of defects detected: Asymptotic Sig. = 0.001 # of delocal defects detected: Asymptotic Sig. = 0.02 # of false positives generated: Asymptotic Sig. = 0.01 Students: no inspection technique out or underperformed any of the other techniques. Industry professionals: no inspection technique out or underperformed any of the other techniques. Industry professional participants performed significantly better than student participants.
Conclusions	There is a significant difference between industry professionals and students in empirical research. There is no difference between inspection techniques for developers of similar experience levels. Different inspection techniques impact experience levels in different ways, professional developers preferred less structure, and student developers preferred more structure.

Table 12-2. Summary of Chapter 6's experiment results.

Experiment Details	Summary
Aims	To measure the cognitive levels developers operate at while implementing a UBR technique mapping a sequence diagram to the underlying code.
Participants	Industry professionals, undergraduate students.
Artefacts	Command line interface Java audio player.
Results	Knowledge and Comprehension accounted for 70% of developers verbalised thinking process.
Conclusions	With no background knowledge of the code using sequence diagram inspections to increase the inspectors' cognitive level is ineffective.

Table 12-3. Summary of Chapter 7's experiment results.

Experiment Details	Summary
Aims	To identify if there is a correlation between the number of defects detected by an inspector and the number of successful modifications that inspector makes to the inspected code.
Participants	Final year students.
Artefacts	A navigation system written in Java.
Results	Correlation between number of defects detected and number of successful modifications $R^2 = 0.32$. Pearson Product-Moment Correlation: r = 0.56, p = 0.09 There is a weak correlation between the number of defects detected and the number of successful modifications a developer makes to the same code base. Two way t-test for the different ways in which inspectors change code compared to non-inspectors: p-values of 0.001 & 0.02 indicating significant differences. Inspectors stated that performing a code inspection increased their understanding of the system.
Conclusions	Carrying out code inspections is an effective method for increasing developer productivity, as the developer is already familiar with the code and how it works. Undertaking a CBR inspection positively affects a developer's ability to make changes to the code. Developers categorise their understanding and ability to modify the code higher than if they had not inspected the code.

Table 12-4. Summary of Chapter 8's experiment results.

Experiment Details	Summary
Aims	To measure the cognitive levels of developers performing Ad hoc, CBR and ADR inspections.
Participants	Final year students.
Artefacts	Java code for the game of Battleship
Results	Kruskal-Wallis test comparing each inspection technique with the different Bloom's Taxonomy levels: no significant difference for Comprehension, Application and Analysis. For Knowledge and Evaluation p-values of 0.004 & 0.006 indicating significant differences. For Ad hoc inspections 60% of inspectors' utterances are in the Knowledge category and only 12% in the Evaluation category. For ADR inspections 50% of inspectors' utterances are in the Knowledge category and 18% in the Evaluation category. For CBR inspections 29% of inspectors' utterances are in the Knowledge category and 39% in the Evaluation category.
Conclusions	CBR is significantly more effective in facilitating developers to operate at higher cognitive levels.

Table 12-5. Summary of Chapter 9's experiment results.

Experiment Details	Summary
Aims	To measure the cognitive levels of developers adding new functionality to a code base after performing an Ad hoc, CBR or ADR inspection on that code base.
Participants	Final year students.
Artefacts	Java code for the game of Battleship.
Results	Carrying out a prior inspection assisted developers to function at higher cognitive levels during this task.
Conclusions	Having previously carried out a software inspection facilitates developers to function at higher cognitive levels while adding new functionality to the inspected code base.

This tabular summary, presenting the results of each experiment, in the context of this thesis, show that software inspections are an excellent way to assist developers to operate at higher cognitive levels. The less experience the developer has, the more structure they need in the implemented reading strategies.

Chapter 10 addressed the research issue of the lack of a mapping between software inspection techniques and the cognitive process models they support developers to implement. It also addressed the issue of the mapping from software inspection techniques to the Bloom's Taxonomy level they facilitated in developers.

The chapter examined and mapped the software inspection techniques used within this thesis to the cognitive process models at which they required inspectors to operate and the Bloom's Taxonomy level that they facilitated in developers. The mappings were created to enable software development teams to know which cognitive process model and which cognitive levels each technique supports developers to operate at.

This knowledge is important in decision-making regarding what techniques to

implement depending on the desired outcome. It is also important when combined with the prior understanding regarding the impact developer experience has on performing a code inspection. Specific reading techniques can be deployed with an understanding as to the cognitive process model it supports, the Bloom's Taxonomy level facilitated, and how much structure is needed by the developer depending on their experience level.

Chapter Eleven made recommendations for guidelines regarding the application of software inspection techniques as a reading strategy to assist with programmer comprehension. Suggestions were also made about introduction and implementation of software inspection techniques as a reading technology into university degrees as well as training programs. The aim is to teach developers the art of reading software code.

Software developers, at some stage of their career, will be required to read code written by another developer. This essential skill needs to become a skill that is explicitly taught rather than one that is caught or learned on the side. The guidelines and recommendations made within Chapter Eleven are based on the work reported within this thesis.

Developers need to be able to read code and read it effectively. Software inspections as a reading strategy to increase the cognitive levels at which developers operate have been demonstrated, within the context of the experiments reported within this thesis, as one means of doing this.

Chapter Eleven's guidelines and suggestions, constructed upon Chapter Ten's mapping, are the culmination of this research. The application of these guidelines present a way in which this technology can be applied in working towards teaching the skills of effective code reading to developers. Increasing software developers' efficacy in reading code can significantly impact their code understanding, leading to an increased ability to write higher quality code as well as reducing the amount of time which they invest in attempting to understand code written by other developers.

12.3 Future Work

While this thesis has provided solutions to the issues described in Chapter Three, there remain outstanding issues within the area of program comprehension that can use the work within this thesis as a foundation.

The results provided within this thesis would benefit and continue to be strengthened through having larger sample sizes to analyse and compare. Larger sample sizes would continue to strengthen the discoveries reported within this thesis, and it would also allow other correlations and relationships, if they exist, to emerge from the data.

Longevity studies would also benefit this research. The tracking of students entering into a degree program where they are introduced to specific reading techniques and then tracked throughout the program in order to correlate how the reading technique impacted upon their learning and code reading skills.

In this thesis, a checklist to assist developers with detecting defects was implemented. Sillito et al. (2008) identified 44 questions that developers asked when

attempting to comprehend and learn a program. The creation of a comprehension checklist may be beneficial. Novice developers expressed that they thought the checklist assisted them in carrying out the inspection task. A comprehension checklist may prove to be a technique that supports developers in their task of comprehending a program.

In the area of developer cognition, measuring cognitive loads, different from cognitive levels that developers experience while carrying out software inspection tasks, may assist in continuing to improve the software reading technologies currently being used within the software development community. Carrying out new studies using Electroencephalography (EEG) technology as a way to measure cognitive loads placed on developers as they perform software inspection activities, may assist in the continued refinements of these technologies in order to better understand the effects that the techniques have upon the brain. The EEG has already been used different areas in computing such as examining the working memory load in neural network pattern recognition (Gevins et al. 1998).

The use of Electroencephalography (EEG) technology can help to initiate some of the next major steps in assisting software developers as they are faced with the exponential use of software as well as exponential increase in the complexity being written into the software systems that have become an integral part of today's society.

12.4 Conclusion

We live in a world that is highly dependent upon software. The financial systems

that keep the world economies alive are software-driven; the aircraft we fly are software-controlled; the energy grids that supply electricity to houses and businesses are operated through software systems; the water supply, the waste management systems, the telephone systems, are all controlled by software. Humans wrote the software that controls these devices and infrastructure. Humans are known to make omissions and mistakes. Within software systems, these omissions and mistakes can be, and are, expensive as well as life threatening in some cases.

In this thesis, strategies and methodologies have been demonstrated that increase the cognitive levels at which software developers operate while carrying out different software engineering tasks. Operating at these higher cognitive levels enables developers to understand and comprehend the system in a more in-depth manner. Understanding and comprehending the system enables the developers to be able to evaluate the system for correctness that is both operational and functional, and then to be able to synthesise corrections and modifications and introduce new functionality into the system in a manner that maintains the system's operating integrity.

The safety and functioning of today's world is more dependent upon this than it was when the first Software Engineering conference was called for in 1968 due to "an awareness of the rapidly increasing importance of computer software systems in many activities of society" (Software Engineering 1968).

Therefore, with demand high for software developers in most areas of business, and the apparent shortage of developers to take up these positions, it is of the utmost

importance that new methodologies and frameworks be created to aid developers, either novice or experienced, new to a project, to better and more quickly understand the system on which they are working.

Bibliography

1968, *Software Engineering,* Naur, P., & Randell, B. eds. NATO Scientific Affairs Division, Garmisch, Germany.

Gevins, A., Smith, M.E., Leong, H., McEvoy, L., Whitfield, S., Du, R. & Rush, G., 1998, Monitoring Working Memory Load during Computer-Based Tasks with EEG Pattern Recognition Methods, *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40(1), pp. 79-91.

Kothari, S.C., 2008, Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, *Scalable Program Comprehension for Analyzing Complex Defects.* pp. 3-4.

Sillito, J., Murphy, G. & De Volder, K., 2008, Asking and Answering Questions During a Programming Change Task, *IEEE Transactions on Software Engineering*, 34(4), pp. 434-51.