A Comparison of Code Inspection Techniques

by

David Andrew McMeekin

Submitted to
the Department of Computing
in partial fulfillment of the requirements for the degree of

Bachelor of Science (Computer Science) (Honours)

at

Curtin University of Technology

November 2005

In memory of

Whitney Faith Steven 15 June 2004 – 22 May 2005.

A Comparison of Code Inspection Techniques

by

David Andrew McMeekin

Submitted to the Department of Computing
in November, 2005 in partial fulfillment of the
requirements for the degree of
Bachelor of Science (Computer Science) (Honours)

Abstract

A software inspection is the process of reading software artefacts to find defects. Since their formalisation as a software engineering practice, software inspections have become a vital part of the software development lifecycle. Inspections were created and perfected when the procedural programming paradigm dominated. Due to the increased usage of object–oriented code, these inspection techniques need further development to cope with the added functionality and complexity inherit within large scale object–oriented systems. Two early empirical studies attempted to address these problems by specifically developing techniques with object–oriented code in mind. The two studies returned contradictory results. Through the design and implementation of two further empirical studies, this research identified that these earlier studies returned contradictory results due to uncontrolled variables. This research found no statistically significant difference between the code inspection techniques as reported by the previous studies. These results reflect the continued need to develop inspection techniques that adapt with the inspector's experience level. By comparing student and industry based participants within this study, it is clear that there is a statistically significant difference in the ability level of these groupings. This calls into question the results and recommendations of previous empirical studies in software engineering where 87% of the participants have been students. It is important for the software engineering research community to continue conducting empirical research, within both university and industry environments to further mature the software engineering discipline.

Acknowledgments

To my supervisors, Dr Brian von Konsky and Dr Mike Robey, your guidance, encouragement and enthusiasm has helped guide me through this year. I so enjoyed working with you both, thank you. Brian, nobody responds faster to email than you, thanks and remember 'Bis zur Unendlichkeit und noch viel weiter...' Mike, your office door is always open, thanks. To David Cooper, your support, encouragement and friendship throughout the year has been greatly appreciated. To the students of Curtin University's Department of Computing who took part in my studies, thank you. To the industry professionals who willingly gave up their time for the study, thank you. Mr Andrew Marriott who not only helped with my ethics approval but was the person who dared me to even consider Honours, thank you. Patrizia Nanni, your support and assistance throughout this year was so appreciated, thank you.

Finally and by far most importantly, my family. To my wife Katharina, you're the best. This year would not have been possible without your love and support: THANK YOU!!! To my children Josiah, Marta and Jozua: Dad can come play, finally.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> A defect is a deviation from the specification. Its precise nature depends on the material being inspected, but it is basically something that would go on to cause failure (some undesired behaviour of either a trivial or catastrophic nature) if left uncorrected (Dunsmore, 2002, p. 148).

A software inspection is the process of reading software artefacts to find defects. The importance of software inspections cannot be underestimated. Today's world is 'dangerously dependent on large software systems whose behaviour is not well understood and which often fail in unpredictable ways' (President's Information Technology Advisory Committee, 1999, p. 4). The estimated annual cost (in the U.S.A) of failing to detect defects in software is $22.9 - 59.5 billion (National Institute of Standards and Technology, 2003, p. ES-3).

This research looks specifically at object–oriented (OO) code inspections. Current inspection techniques were designed, developed and perfected while the procedural programming technique was most dominant. OO usage has increased in the past 15 years. Inspection techniques need to be further developed to cope with these changes.

1.1 Background to the Aims

Thelin *et al.* and Dunsmore *et al.* conducted separate studies on inspection techniques specifically designed for OO code.

Thelin *et al.* compared Usage-Based Reading (UBR) with Checklist-Based Reading (CBR). In UBR the inspector is guided through the code by prioritised use cases [1] (see section 2.2.9). In CBR the inspector is guided through the code by a checklist that asks a series of questions (see section 2.2.1).

Dunsmore *et al.* compared CBR with Use-Case Reading (UCR) and Abstraction-Driven Reading (ADR). In UCR the inspector is guided through the code by non-prioritised use cases (see section 2.2.8). In ADR the inspector systematically reads the code and writes a natural language abstraction about each method (see section 2.2.10).

Thelin *et al.* reported positive results both in efficiency and effectiveness for UBR over CBR in finding faults most critical to users (Thelin *et al.*, 2003, p. 687). Dunsmore *et al.* reported weak results for the UCR technique in comparison to both CBR and ADR (Dunsmore *et al.*, 2003, p. 685).

The UBR and UCR techniques differ in one aspect, UBR prioritises the use cases and UCR does not. These two studies reported seemingly contradictory results. Dunsmore *et al.* (2003) reports UCR as the worst while Thelin *et al.* (2003) report UBR as the most effective.

Denger *et al.* followed up the work of Dunsmore *et al.* (2003) and Thelin *et al.* (2003) with another empirical study. The results were inconclusive (Denger *et al.*, 2004).

[1] A use case captures a contract between the stakeholders of a system about its behaviour. The use case describes the system behaviour under various conditions as it responds to a request from one of the stakeholders (Cockburn, 2000, p. 1)

These three empirical studies form the basis for further exploration of OO code inspection techniques.

Sjoberg *et al.* (2005) show that on average, 87% of empirical study participants are students and 9% industry professionals. They state that 'within software engineering empirical studies, the low use of professionals may inhibit the understanding of industrial software processes and, consequently, technology transfer from the research community to industry' (Sjoberg *et al.*, 2005, p. 739). In analysing the results of the empirical studies performed, this study also takes Sjoberg *et al.*'s work into consideration.

1.2 Aims

This research extends the studies of Thelin *et al.* (2003) and Dunsmore *et al.* (2003) in comparing the UBR, UCR and CBR techniques in order to understand the apparently contradictory results reported. The aims of this research project were as follows:

1. To understand the discrepancies between the results of Thelin *et al.* (2003) and Dunsmore *et al.* (2003). Insight into the differences between their results will facilitate further advancements in inspection techniques for OO code.
2. To compare the OO code inspection techniques (UCR, CBR and UBR) to determine which technique detects the highest number of defects within a given period of time.
3. To make recommendations regarding effective OO code inspection techniques.

1.3 Outcomes

Successful achievement of these research aims will have two major outcomes:

1. recommendations, based upon objective data, regarding effective techniques for the

inspection of OO code, in order to maximise defect finding within a workable time frame, based on objective data; and

2. recommendations regarding inspection techniques that require further investigation and research in order to continue to advance the efficacy of inspection techniques in Outcome 1.

1.4 Significance

This research will contribute to the discipline of software engineering to develop better inspection techniques for OO code and hence to better improve software quality.

1.5 Ethics

This research involved people in organised empirical studies. Under the Curtin University Human Research Ethics Committee guidelines, this research was classified as a 'minimal risk' project. 'Minimal Risk' projects are those where 'the probability and magnitude of harm or discomfort anticipated in the research are not greater in and of themselves than those ordinarily encountered in daily life' (Curtin University of Technology, 2005).

Approval of the first study was granted on 16 May 2005, ethics approval number ESC-02-2005. An extension of this approval was sought for the second study (due to changes in the artefacts) and was granted on 23 August 2005 under the same ethics approval number. All collected data has been made anonymous and the identities of participants cannot be recovered by anyone.

1.6 Thesis Structure

This thesis is structured as follows.

Chapter 2 provides a background to software inspections. It looks at the history of code inspections, when they were first performed, how they developed and when they were formalised. It then introduces the most commonly used inspection techniques and gives a brief usage description.

Chapter 3 looks at the nature of OO programming and its impact on software inspection techniques. It examines some OO features describing their impact on the inspection process.

Chapter 4 explains the methodology and approach used within this research. It lays out the procedures of the two empirical study's conducted along with the studys limitations and threats to their validity.

Chapters 5 and 6 report on empirical studies one and two respectively.

Chapter 7 discusses the results of the empirical studies within the context of the research aims listed in section 1.2. It also presents the recommendations arising from these results, and possibilities for future work.

Chapter 2

Background

> 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 able to check one's own work, but also to have one's work checked and to be able to check the work of others (Petroski, 1992, p. 52).

2.1 Inspections

The word inspection is defined by the ANSI/IEEE Std. 729-1983 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 (Standards Coordinating Committee of the Computer Society of the IEEE, 1983).

Code inspections have been in existence for as long as software has existed. Charles Babbage had Ada Lovelace and others inspect his programs. John von Neumann willingly

gave his programs to colleagues for review. In today's standards these would be known as informal inspections. Nevertheless, they were still inspections. By the end of the 1950s, software project managers began to see the need for formal inspections, and in the 1960s inspections began to become formalised (Weinberg and Freedman, 1984, p. 72).

Inspections of design and code were first formalised as a software engineering process in 1976 when Fagan published 'Design and code inspections to reduce errors in program development' (Fagan, 1976). Since then, inspections have become a normal, integral and expected part of the software development lifecycle. Fagan (1976) clearly shows that source code and design document inspections have improved the discovery and subsequent removal of defects, and have therefore improved the quality of both the design and the code. 'Inspections help reduce defects in a software system by ensuring that the software artefacts which are necessary for its construction correctly reflect the needs of the stakeholders' (Travassos *et al.*, 1999, p. 47). Code inspection techniques are used to ensure that the constructed product performs and functions in a manner that correctly fulfils the needs of the stakeholders.

Fagan (1976) demonstrates that defects discovered late in the software development life cycle require significant rework, sometimes costing between 10 and 100 times more than the initial project. The closer to its introduction that a defect can be discovered, the easier and less expensive it is to remove. Fagan (1976) also showed that up to 85% of errors were detected through inspection, resulting in a 25% saving in programmer time in some cases (Fagan, 1976, pp. 185–194). Fagan (Fagan, 1986, p. 745) also reported that the application of inspections yielded a discovery of 50 - 93% of all defects, with more than an 85% saving in programmer time.

IBM awarded Fagan their highest technical award, its Outstanding Contribution Award, for his work on inspections (Gilb and Graham, 1993, p. vii). The relevance of Fagan's original 1976 article was such that it was republished in the *IBM Systems Journal* 23 years later, without change (Fagan, 1999). Also noteworthy is that between 1976 and 1999, very few important improvements were made to code inspection techniques, despite

Role	Description
Moderator	Moderates and manages the inspection team
Designer	The person who produced the program design
Coder Implementer	The person who translated the design into code
Tester	The person responsible for writing and/or executing the test cases

Table 2.1: The four roles

the fact that the introduction of OO techniques resulted in a change in programming paradigms.

Inspection techniques, also known as reading techniques, are step-by-step procedures that guide individual inspectors as they uncover defects in a software artefact. These techniques enable systematic and well-defined strategies for inspecting a document, allowing for both feedback and improvement (Shull *et al.*, 2001, p. 726).

Inspections are defined as 'a formal, efficient, and economical method of finding errors in design and code' (Fagan, 1976, p. 189). A formal inspection team generally contains four people each playing a specific role within the inspection process. This is shown in Table 2.1 (Fagan, 1976, p. 190).

Fagan's inspection process was a five step procedure, as described below (Fagan, 1976, pp. 192–194):

1. **Overview**: the program designer gives a general overview of the area being addressed to the entire inspection team.
2. **Preparation**: individual inspectors look at the artefacts and attempt to understand the details of what the code should be doing. Checklists created from defects detected in earlier inspections are used to guide the inspectors.
3. **Inspection**: the team meets as a group and attempts to discover as many defects as possible.
4. **Rework**: the defects discovered in the inspection are resolved by the coder/implementer.

5. **Follow-up**: it is the responsibility of the moderator to ensure that the designer and coder/implementer resolve all defects discovered in the inspection.

Over the past 30 years the 'Fagan Inspection' has become a standard process within many software engineering teams (Tyran and George, 2002, p. 87).

2.2 Inspection Techniques

A reading technique is 'a series of steps or procedures whose purpose is for an inspector to acquire a deep understanding of the inspected software product.' It can also be considered as a tool used by inspectors to detect defects within the product under inspection (Laitenberger and DeBaud, 2000, p. 13).

This section examines several different inspection or reading techniques.

2.2.1 Checklist Based Reading

Checklist Based Reading (CBR) is considered the standard inspection technique used in software organizations around the world (Laitenberger and DeBaud, 2000) and was first recommended by Fagan (1976). Each type of document to be inspected has its own checklist.

As CBR is considered the standard inspection technique in use today, it forms a baseline from which this research was conducted.

Brykczynski (1999) surveyed 117 checklists from 24 different sources and came up with five commonly suggested heuristics for the creation of an effective checklist.

		Y	N
Methods	1. Are all parameter names consistent? 2. Are all parameters used within the method?		

Table 2.2: A section of a sample checklist.

1. Checklists need to be updated on a regular basis to encourage inspectors to read and use them.
2. Checklists should be a single page in length, preventing the need to turn pages while inspecting. It is then possible for the checklist to sit on the desk of the inspector as the code is inspected.
3. Checklist items should be worded in question form so that answering 'no' to the question would indicate a defect.
4. Checklist items should not be vague.
5. Checklist items should not be used in places where other methods are better used to enforce that which is needed or required.

Table 2.2 displays two questions that may be found in a 'Methods' section of a checklist.

2.2.2 Ad-hoc

The Ad-hoc technique is one of the simplest techniques currently used. This technique provides no formal guidance to the inspector regarding how the inspection should be implemented or what they are looking for. Although considered simple, it is very effective. There are no formal instructions or directions given to the inspector. It is assumed that the inspector will still carry out a thorough and systematic inspection of the software, using personal experience and understanding to guide the inspection (Laitenberger and DeBaud, 2000, p. 13).

A strength of this method is that it provides the greatest freedom for the experienced inspector to find defects located within the software (Laitenberger and DeBaud, 2000,

p. 13) (Dunsmore, 2002, p. 10). However, the lack of a formal process may also mean that an ad-hoc inspection carried out by an inexperienced inspector may not identify many defects.

2.2.3 *N*-Fold Inspections

The N-Fold inspection technique was first introduced by Martin and Tsai (1990). Inspections are conducted in parallel by N different teams. A moderator supervises the team inspections. The moderator collects the results from each team, collates them and a new version of the artefact is generated (Martin and Tsai, 1990, p. 226).

'The N-fold inspection method is based on the hypothesis that the N separate inspection teams do not significantly duplicate each others' efforts and that there is not a high degree of fault-detection overlap' (Schneider *et al.*, 1992, p. 190). For example, three teams perform an inspection on two classes. A combined result of the three teams will produce a significantly better result, with a minimal number of common defects being found. This then justifies the cost of the N, or in this case, the three fold inspection.

2.2.4 Phased Inspections

Phased inspections, designed by Knight and Myers (1993), attempted to broaden the concept of inspections. Until that point in time, inspections were mainly focused upon finding defects within the artefact being inspected (Fagan, 1976; Parnas and Weiss, 1985; Fagan, 1986; Ackerman *et al.*, 1989). However, Knight and Myers wanted inspections to also look at maintainability, portability and reusability issues associated with the artefact being inspected. Their basis for this was that although a software product may be defect free, its value may be lessened by the fact that it is not easily maintainable (Knight and Myers, 1993, p. 52).

This type of inspection contains a series of partial inspections, each known as a phase. The artefact that is inspected needs to be easily comprehensible. If not, then the artefact size must be reduced. For example, suppose an inheritance hierarchy is to be inspected, which contains over 150 different classes. To inspect 150 classes at one meeting may be impossible. To make it more easily comprehensible, a base class may be chosen and only classes designed to inherit from it are inspected. This reduces the complexity of the inspection task.

During the inspection either a single property (as demonstrated by the above example) or several small tightly coupled properties are examined. This technique allows subsequent inspections to be based upon the assumption that artefacts that have already been inspected are correct and free from defects. The inspectors of each phase are held responsible for artefacts they have inspected. Their agreement to move onto the next phase suggests that all that has been inspected is now correct (Knight and Myers, 1993, p. 53).

2.2.5 Scenario-Based Reading

Scenario-Based Reading (SBR) was first introduced by Porter and Votta (1994) and is based upon the Ad-hoc and CBR techniques. The Ad-hoc technique is considered to be unstructured, while CBR provides more structure to the inspectors. SBR provides even more structure and guidance to the inspectors. It does so by the use of scenarios which are 'collections of procedures for detecting particular classes of faults.' Multiple scenarios are executed. Each inspector executes a single scenario, and multiple inspectors are coordinated to achieve broad coverage of the document (Porter *et al.*, 1995, p. 564). Porter *et al.*'s reasoning for this method was that 'systematic approaches with specific, coordinated responsibilities reduce gaps, thereby increasing the overall effectiveness of the inspection' (Porter *et al.*, 1995, p. 564).

2.2.6 Perspective-Based Reading

Perspective-Based Reading (PBR) was originally designed for use with requirements documents (Basili *et al.*, 1995). It has also been used within the context of code inspections (Laitenberger and DeBaud, 1997; Laitenberger *et al.*, 2001). PBR's goal is 'to read a software artefact from the perspectives of the artefact's various customers for the purpose of identifying defects' (Laitenberger and Atkinson, 1999, p. 495). Examples of the customer include the designer, tester or maintainer.

In PBR, each inspector is guided by a scenario. This scenario is an algorithmic guideline that shows the inspector how they should proceed while reading the artefact. Using this scenario should result in an in-depth understanding of the document (Laitenberger and DeBaud, 1997, p. 781).

2.2.7 Sample-Driven Inspections

A Sample-Driven Inspection (SDI) is one in which only a sample of the software artefacts are inspected. It is an alternative to complete inspection. This technique analyses a sample of the defects in a certain number of pages (Gilb and Graham, 1993, pp. 75–76). This allows a defect density to be calculated, and results in an estimate of the number of remaining defects. A key assumption is that the mistakes made in the sampled group of pages will be the same as those made throughout the remaining pages.

One motivation behind SDI is to reduce the effort in an inspection session while guiding the inspection to focus on the artefact(s) with the most defects. The SDI process contains two parts. The first part is where samples are looked at to determine the artefacts with the most defects. The second part is a more complete inspection of those artefacts with the most defects (Thelin *et al.*, 2001a, p. 81).

There is no specific reading technique used for this method and reviewers do not prepare

in any special manner. For the inspection itself, inspectors simply choose their preferred technique (Thelin *et al.*, 2001a, p. 81).

2.2.8 Use Case Reading

The Use Case Reading (UCR) technique was designed to assist with the inspection of OO systems (Dunsmore *et al.*, 2003). The goal is to ensure that every object will respond in a correct manner to the possible ways in which it will be used within the system. That is, the correct methods are called and changes of state are consistent and correct (Dunsmore *et al.*, 2003, p. 680).

In UCR, inspectors create several different scenarios from the use cases and test how these scenarios are realised by the class under inspection. Implementing this technique requires the context in which the object is being used to also be examined. It is expected that incorrect, missing or error-prone method calls will be discovered using this technique. It is possible that some methods within a class will not be examined, as they are not required within the context of the created scenario. These methods must therefore be examined using a different technique (Dunsmore *et al.*, 2003, p. 680).

The scenarios that the inspector creates are used to trace the message calls between objects in the sequence diagrams[1]. When the class under inspection is encountered in the sequence diagram, the inspector changes artefacts and inspects the code, verifying the expected behaviour. Upon completion of the inspection, the object's state should match the inspector's expectation. If not, this suggests there is a defect within the code (Dunsmore *et al.*, 2003, p. 680).

[1] A sequence diagram from the UML specification 'is an interaction diagram that emphasizes the time–ordering of messages' (Booch *et al.*, 1999, p. 25).

2.2.9 Usage-Based Reading

Usage-Based Reading (UBR) is a technique developed from Statistical Usage Inspection (SUI) (Olofsson and Wennberg, 1996). The SUI technique focuses on the certification of the reliability of a product in question by testing it according to the expected usage of the system (Olofsson and Wennberg, 1996, p. 21).

An inspection's success is determined by the number of faults detected, irrelevant of the impact they may have on the final system quality. It is therefore important that the inspection detects the most critical defects (Thelin *et al.*, 2003, p. 690). The UBR technique focuses upon this. All faults are not considered equal within this technique. Defects that have the most damaging impact upon the user's interaction with the system are deemed the most important. This reading technique is controlled by a prioritisation of the use cases. This prioritisation is based on the perceived importance of the system's functionality from the user's (or group of users) perspective (Thelin *et al.*, 2003, p. 690).

Once use cases have been prioritised, the inspection is conducted in a similar manner to the UCR technique (see section 2.2.8).

2.2.10 Abstraction-Driven Reading

The Abstraction-Driven Reading (ADR) technique was created specifically with OO code in mind. The code is read in a systematic way and a natural language abstract specification is written by the inspector. In this way, an inspector's understanding of the code is increased as they follow method calls and other outside information such as the original specification, as needed (Dunsmore, 2000a, p. 3). An advantage of these natural language specifications is that 'any future inspections that encounter references to methods for which an abstraction exists need only read the abstraction' (Dunsmore *et al.*, 2003, p. 680).

The technique begins with an analysis of the system-wide coupling between classes. Those

with the lowest number of couplings are inspected first. An inspection of the methods within those classes then occurs, with an analysis of the couplings within those methods. In the same manner, the methods with the lowest couplings are inspected first. As the inspection proceeds, the number of abstractions increases, reducing the amount of time the inspector spends looking for information not located within the method or class being inspected (Dunsmore *et al.*, 2003, p. 681).

2.2.11 Traceability-Based Reading

Traceability-Based Reading (TBR) was developed specifically for high-level OO design documents created using the Unified Modeling Language. This technique explicitly looks at the high level design documents such as class diagrams, sequence and collaboration diagrams, state machine diagrams and package diagrams (Travassos *et al.*, 1999, p. 48). The purpose of this technique is for design documents to be clear and free from ambiguity. When the low level design is started it allows the developers to implement a solution to the presented problem, with as few defects as possible. Although TBR is a reading technique, it is aimed at higher level software artefacts than those for code inspections. This research looks specifically at inspection techniques aimed at code, and will not therefore explore this technique any further.

2.3 Summary

This section presented an overview of inspection techniques currently used within the software industry. These techniques have one clear goal in mind: to deliver high quality software products.

The next section examines the challenges that OO poses to inspection techniques.

Chapter 3

Object–Orientation and Delocalisation

> Zero–defect software is the Holy Grail of all software developers. It has proved to be an elusive goal... (Gilb and Graham, 1993).

Design and code inspections are a vital part of the software development life cycle. Ideally every piece of software should be defect free at first release. Unfortunately, this is rarely the case. The introduction and development of inspections (as described in Chapter 2) is aimed at moving closer to the 'Holy Grail.' Historical data and numerous case studies indicate that inspections have increased software quality over the past 25 years. With the increased usage of the OO paradigm, this same goal remains: zero–defect software.

The majority of inspection techniques currently in use were developed and perfected while the procedural programming paradigm was predominant. The techniques listed in Chapter 2 clearly demonstrate this: four of the 11 techniques were designed for OO software systems. Over the past 15 years, the OO programming paradigm has become much more prominent. The changes brought about by this paradigm shift have uncovered the inadequacies of the current inspection methods when applied to OO software artefacts (Dunsmore *et al.*, 2003, p. 677).

3.1 Object–Oreintation

The introduction of Object–Oriented Programming (OOP) was welcomed in industry because of the many features it provided that were not available within the procedural programming paradigm, such as function and operator overloading[1], inheritance, dynamic (late) binding and polymorphism (Lejter *et al.*, 1992, p. 1045) (Wilde and Huitt, 1992, p. 1038). Booch expected that through OOP higher quality software would be produced due to these features providing better data abstraction and information hiding, allowing more readily for concurrency, and responding better to changes in the real world (Booch, 1986). It soon became evident, though that these advantages may also cause problems maintaining OO designed systems (Wilde and Huitt, 1992; Lejter *et al.*, 1992; Wilde *et al.*, 1993).

At the core of software inspections is an in-depth understanding of the code. These new OOP features greatly increased software code complexity, requiring more time for an in-depth understanding.

Wilde *et al.* describe OOP and its influence upon program understanding in this manner:

> The basic Object–Oriented strategies of dividing the world into object classes, encapsulating object internals using message passing, limiting any one object's responsibilities, and reusing objects in more than one context lead to a profusion of relatively small program parts with many potentially complex relationships. The different appearance and organisation of object-oriented code may startle many programmers, and their traditional approaches to program understanding may break down (Wilde *et al.*, 1993, p. 76).

The problems that OOP introduced to system maintenance are also relevant to software inspections. The ability to read, question and understand what a program is doing is

[1] these were available within Ada.

essential to successful inspections. The following section explains the OOP principles, their effects program complexity and, an inspector's ability to understand the OO program and then search for defects within it.

3.2 Delocalisation

Delocalisation arises when 'pieces of code that are conceptually related are physically located in non-contiguous parts of a program' (Soloway *et al.*, 1988, p. 1261). Delocalisation is not unique to OOP. It also existed in the procedural programming paradigm through function calls (function calls made it more difficult to read code from top to bottom). However, the features of the OO paradigm greatly increased the problem of delocalisation (Dunsmore *et al.*, 2003, p. 677).

A program written in a delocalised manner may be difficult to read and understand. In the case of OO programs, the reading and understanding of a single class may become difficult because not all the code needed to understand a specific class is located within it. This affects the inspector, causing them to make assumptions about the delocalised code. Although the assumptions may be correct, the implementation of the delocalised code may be incorrect. This may cause the code currently under inspection to be defective and remain this way due to the assumption made by the inspector.

The following piece of Java™ code demonstrates the effect of delocalisation within a code inspection. This code was used by Dunsmore *et al.* (2000b) in an experiment conducted and published in a subsequent conference proceeding (Dunsmore *et al.*, 2000a, p. 468).

```
private void purge()
{
    GregorianCalendar today = new GregorianCalendar();
    today.roll(Calendar.DATE,false);
```

```
    for(int i=0; i<reservations.size(); i++)
    {
        if(today.after((Reservation)reservations.elementAt(i)))
        {
            reservations.removeElementAt(i);
            date = 0;
        }
    }
}
```

The `purge()` method is part of video library system. The method checks the elements of the `reservations Vector` for any out of date reservation. The date from the currently referred to reservation should be passed to the `GregorianCalendar after()` method. The reservation itself was passed instead. The call to the `getDate()` method was missing from the argument. This would have returned the date of the current reservation.

When compiled, there were no errors and the code could be executed in that state. The `purge()` method is part of the `Reservation` class which is part of the library system being developed. The `GregorianCalendar` and `Vector` classes belong to the Java™ class library.

Understanding this defect requires gathering information from a variety of delocalised sources:

1. The `Vector` class method `elementAt()` is invoked to retrieve the current reservation.
2. The `GregorianCalendar` method `after()` was called for comparing two dates. The compiler did not throw an error with the incorrect parameter being passed to `after()` because the method can receive any object of type `Object` as a parameter. Every object in Java™ inherits from the base class `Object`. The `Reservation` object, via inheritance, is an `Object` type.

3. The `GregorianCalendar` class is a descendant class of `Calendar` where much of its functionality comes from.
4. A typecasting of the object retrieved occurs by use of the `Reservation` class.

(Dunsmore *et al.*, 2000a, p. 468)

The six executable lines of code within the `purge()` method seemed simple enough. However, the points discussed above must be understood by the inspector to comprehend what the code does. The corrected line of code for the innermost branch is:

```
if(today.after(((Reservation)reservations.elementAt(i).getDate()))
```

For a larger set of collaborating classes, it is clear that delocalisation poses a great problem to the inspector's ability to effectively understand the software system and therefore to perform a successful inspection upon it.

The following sections describe in more detail the features of OOP that lead to delocalisation and the effects they have upon code inspections.

3.2.1 Overloading

In the procedural programming paradigm, a function name uniquely identified it. During inspection, when this function is called the inspector moves directly to it and examines it for correctness. An inspector could start at the top of a procedural program and step through the code, tracing each method invocation and checking for correctness. The implementation of overloading within OOP means this is no longer possible.

The way overloading is used within OOP means a method name no longer uniquely identify a method within the system. It is possible that each method name has been used many times throughout the system. The implementation of the method being inspected is not obvious to the inspector in a static inspection. The method must now be looked at within

the context of the system and the entire class hierarchy.

Familiarity with a method that has been implemented and overridden many times may cause the maintainer or inspector to make an assumption about the method which in this specific instance is not correct (Wilde *et al.*, 1993, p. 78).

3.2.2 Dynamic Binding

Dynamic binding adds complexity to the understanding of the code under inspection by binding messages with specific methods at run–time. This makes it impossible for the inspector to statically determine the actual method body that will be called at a specific point in the program (Wilde and Huitt, 1992, p. 1038), (Lejter *et al.*, 1992, p. 1046).

For example: an object O statically declared to be of type T, may be bound to an object of type T or any of its descendent classes. When a class method m is called in object O, the run-time system determines which of the overloaded methods are actually invoked (Lejter *et al.*, 1992, p. 1046).

The inability to know in the static state which m is invoked can prevent a thorough understanding of the code by an inspector who was not part of the development team. A lesser understanding of the code may reduce the inspection's effectiveness, leaving undetected defects within the code.

3.2.3 Inheritance

Inheritance is a commonly used feature of OO languages. 'The fundamental idea of inheritance is that new software elements may be defined as extensions of previously defined ones: existing elements do not have to be modified when used as a basis for new definitions' (Meyer, 1986, p. 395). A single class description may then be a distributed description

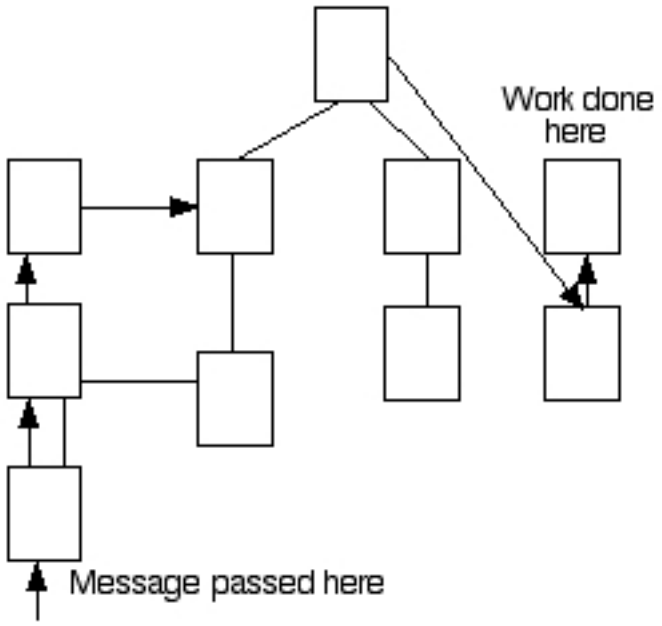

Figure 3-1: A method invocation spread throughout a class hierarchy. Adapted from (Dunsmore, 2002).

with elements of the description being located in other classes. An examination of class C must also contain an examination of all of C's superclasses. In the case of languages that permit multiple inheritance, several class hierarchies may need to be examined to understand the functionality of C in its entirety.

Inheritance clearly demonstrates delocalisation and the problems it causes for an inspector. The inspection of a single line of code within a class may require the inspection of multiple classes spread throughout several multiple hierarchies. This is a time consuming and error–prone process for the inspector. Figure 3-1 depicts this in diagrammatic form.

Traditional inspection methods were not developed to cater for this programming style.

3.2.4 Small Methods

The technique of writing small methods within OOP increases the number of relationships that must be understood by the inspector. The more relationships that must be understood, the greater the complexity of the code. An increase in complexity may lead to a decrease in the number of defects discovered.

Wilde and Huitt also found that for many tasks these small methods had been written with the purpose of 'passing through' the message to another method with little or no processing. This lead to systems having large numbers of very small modules, where traditionally there had been larger methods with a smaller number of modules (Wilde and Huitt, 1992, p. 1040).

A review of inspection techniques currently in use shows that they were not designed to cater for this type of programming.

3.3 Summary

This chapter described the problems some OOP features have introduced to the effectiveness of code inspections. For OO code inspections to produce results similar to those that were seen within the procedural programming paradigm, the existing inspection techniques need to be modified, or new inspection techniques developed. Failure to do so may result in lower quality software systems being developed through the use of OO.

Chapter 4

Methodology, Approach and Limitations

> Research is a 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 (Leedy and Ormrod, 2005, p. 2).

4.1 Methodology and Approach

The research aim was *to understand the discrepancies between the results of Thelin* et al. *(2003) and Dunsmore* et al.*'s (2003) studies.* To do this, two empirical studies were performed.

Thelin *et al.* (2003) compared the CBR technique with the UBR (prioritised use cases) technique and reported that the UBR technique was shown to be more effective and efficient than the CBR technique in their study.

Dunsmore *et al.* (2003) compared a CBR technique with the UCR and ADR techniques, and reported the Use Case technique as the worst performing technique of the three.

The inconsistencies in the results may be due to two things: first, Thelin *et al.* prioritised the use cases inspected. Second, the participants in Thelin *et al.*'s study were masters students with considerable industry experience. Dunsmore *et al.*'s participants were third year honours students with little or no industry experience.

Two goals within the empirical studies designed for this research were to control the reading technique and participant experience level variables. In this study, a composite checklist was developed, combining the salient features of the checklists originally developed by Dunsmore *et al.* (2003) and Thelin *et al.* (2003). The two independent variables, inspection technique and participant experience level were controlled within both studies conducted for this research.

The design and code artefacts used in these studies were also used by Dunsmore *et al.* (2003). These artefacts are well documented, have been tested within earlier studies and were a subset from a much larger set of artefacts. Thelin *et al.*'s (2003) artefacts are equally well documented and tested and, could also have been used to reflect similar results. An arbitrary decision was made to use Dunsmore *et al.*'s.

Approximately 10% of the first empirical study's code was altered to comply with the Curtin University of Technology Java^{TM} coding standard (Curtin University of Technology, 2003). This was to prevent students from being distracted by what may have been perceived as a defect but in reality was only a different coding standard interpretation or implementation.

Participants from industry were recruited in order to allow the results to be generalised to both students and industry.

4.2 The Empirical Studies

Two empirical studies were conducted. The first study controlled the reading technique variable. The second study controlled the participants experience level variable. The two studies were completely independent, and participation in the first did not exclude participation from the second.

4.2.1 Defect

Within these studies a defect was defined as 'a deviation from the specification that would go on to cause failure (some undesired behaviour of either a trivial of catastrophic nature) if left uncorrected' (Dunsmore, 2002, p. 148).

4.2.2 The Code

Java™ code was chosen for the inspections as it was the language used by Dunsmore *et al.* (2003). Student participants would come from the Department of Computing at Curtin University. Java™ is a language that all student participants are familiar with. It is taught within two introductory units, Software Technology 151 and Software Technology 152, in the Bachelor of Science (Computer Science), the Bachelor of Science (Information Technology) and the Bachelor of Engineering (Software Engineering) degrees.

All industry participants were asked if they had a working knowledge of Java™ and a general understanding of UML prior to taking part in the study. Exclusion from participation would have occurred if these two requirements were not met, but this was never the case.

4.2.3 Seeding of Defects

The defects within the code were seeded by Dunsmore *et al.*. This seeding was influenced by the following factors: data collected by Dunsmore *et al.* from two earlier experiments, an industrial survey and Dunsmore *et al.*'s own literature survey and common mistakes made during coding. The seeded defects within the two studies were both delocalised and localised in nature (Dunsmore, 2002, 2000b).

4.2.4 The Participants

There were two participant groupings within these studies. Participants in the first grouping were students, and those in the second were practicing IT professionals from local industry. Participation was voluntary. Studies were not part of any unit and therefore not linked to any assessment.

For both empirical studies, all student participants had either completed or were enrolled in third year Computer Science, Information Technology or Software Engineering, or fourth year Software Engineering or Computer Science Honours degree programs. All students had also successfully completed Software Engineering 252/552 or equivalent. This enabled the independent variable of experience to be controlled.

4.2.5 Commonalities Between Studies

All participants within the two studies were given the following artefacts related to the inspection:

1. An information sheet explaining the purpose of the study.
2. A Consent To Participate Form.

3. An initial natural language specification of the software system.
4. A class specification.
5. An entire system class diagram.
6. The Java^{TM} code of the classes to be inspected.
7. A demographics form about themselves.
8. Defect reporting form.
9. Feedback form.

Participants had no prior knowledge about the system to be inspected before commencing the inspections. All participants were instructed to read the natural language specification first and then the class specifications to gain a general understanding of the system and its functionality. Once an understanding had been established, participants commenced the code inspection using the technique they had been assigned. Participants were given up to 120 minutes to perform the inspection.

Participants worked alone and were not permitted to discuss the inspection with other participants during the study. Participants noted their inspection commencement and completion times. When a defect was discovered, participants noted the time at which it was discovered and described the defect in their own words. Participants were not required to provide the correction. Participants were simply asked to identify the defect.

Participants were informed that the code had been compiled and executed. Participants were asked not to compile and run the code themselves.

Upon completion of the inspection, student participants were asked not to discuss the study with others. This was because doing so might have affected the results should others choose to take part on a different day.

Participants were asked to fill out a feedback form at the end of the inspection. The purpose was to collect participant perspectives regarding how the inspection could be altered to improve its overall success in finding defects.

4.2.6 Checklist-Based Reading Technique

Participants in the CBR group in both studies were given a checklist to guide their reading of the classes. For each class being inspected, participants were to answer each question on the checklist. An answer of 'yes' suggested there was no defect. An answer of 'no' indicated a defect.

An example from empirical study one: Question 16 on the checklist asked: '*Is the correct method being called on the correct object (including casting)?*' When the inspector reached the line of code `removeAllElements()` within the `UserCollection` class their answer to this question should have been no. This highlighted the possibility of a defect. A closer examination should have revealed the wrong method being called. This was to be recorded in the defect reporting form.

4.2.7 Use Case Reading Technique

Participants in the UCR group in both studies were provided with a set of use case scenarios and sequence diagrams. Participants used the use case scenarios to guide the reading of the sequence diagrams. When the sequence diagram made reference to the code under inspection, participants switched to the code and inspected it for correctness.

An example from empirical study one: Table 4.1 gives the flow of events for the 'Delete User' Happy Days scenario. As the participant read through this flow of events, they were asked to trace it on the sequence diagram while referring to the code. If these artefacts differed, it indicated the possible presence of a defect.

Figure 4-1 shows a section of the sequence diagram. Step 5 of the flow of events should result in a call within the `UserCollection` class to `removeElementAt( index )`. The call in the code was to the `removeAllElements()` method. This resulted in all elements within the `Vector` being removed. Participants should have detected a defect and recorded it on

Flow of Events:	
1.	Use case begins when the user selects Delete User option
2.	System prompts user for user name
3.	System verifies the user name is valid
4.	System checks that there are no borrowed items
5.	Use case ends when the system updates records by deleting the user and all details

Table 4.1: Delete user use case flow of events

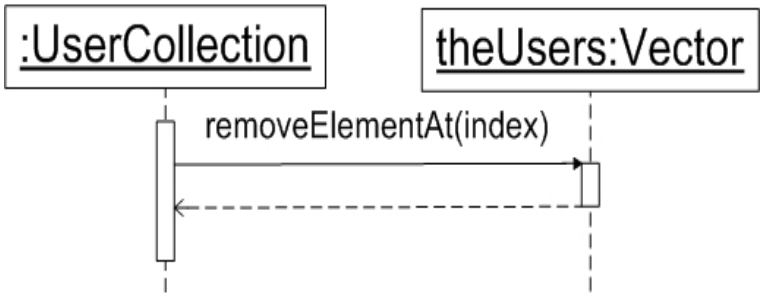

Figure 4-1: Part of the Delete User sequence diagram.

the defect reporting form.

4.2.8 Usage-Based Reading Technique

Participants in the UBR group (in empirical study two) were given the identical information as those in the use case based group. Participants were also told of the use case prioritisation and informed that the prioritised use case should be inspected before any others. The method for inspecting the code was the same as with the UCR group. Participants continued in this manner until all use cases and their extensions were completed.

4.3 Empirical Study One

4.3.1 The Purpose

The goal of this study was to compare the two inspection techniques, CBR and UCR by measuring the number of defects found in a given time period. This study controlled the reading inspection technique variable. This supported the main aim of this research *to understand the discrepancies between the results of Thelin* et al. *(2003) and Dunsmore* et al.*'s (2003) studies.*

4.3.2 The Study

Participants within this study were separated into two groups. The first group performed a code inspection using the Checklist-Based Reading technique. The second group performed a code inspection using the Use-Case Reading technique.

Participants inspected four classes containing 146 effective lines of code, as measured by the program Resource Standard Metrics (RSM). This number of effective lines of code falls within that recommended for a 120-minute inspection period (Fagan, 1976), (Sommerville, 2001, p. 430), (Gilb and Graham, 1993, pp. 78–80). The classes being inspected contained 18 seeded defects. Four of these were defects due to delocalisation.

4.4 Empirical Study Two

4.4.1 The Purpose

The goal of this study was to compare the three independent inspection techniques, UBR, CBR and UCR, by measuring the number of defects found in a given time period. This study controlled the participant's experience level variable as well as the reading technique. This supported the main aim of this research *to understand the discrepancies between the results of Thelin* et al. *(2003) and Dunsmore* et al.*'s (2003) studies.*

4.4.2 The Study

This study consisted of two independent participant groupings. The first group were students. The second were industry practicing IT professionals. All student participants met the criteria stated in section 4.2.4. Industry based participants were required to be active software developers with some working knowledge of Java^{TM} and UML.

The two separate groupings, students and industry professionals, allowed the participant experience variable to be controlled.

The student group was divided into three subgroups. The first subgroup performed a UBR inspection. The second subgroup performed a CBR inspection. The third subgroup performed a UCR inspection.

Due to the low number of industry professionals (18) available for this study, the group was divided into two subgroups. The first subgroup performed a UCR inspection. The second sub–group performed a CBR inspection.

Participants inspected two classes containing 91 effective lines of code, as measured by the

program Resource Standard Metrics (RSM). This number of effective lines of code falls within that recommended for a 120-minute inspection period (Fagan, 1976), (Sommerville, 2001, p. 430), (Gilb and Graham, 1993, pp. 78–80). The classes being inspected contained 14 seeded defects. Eight of those were defects due to delocalisation. Participants also had access to four other classes that were part of the system. The classes belonged to an airline ticketing reservation system.

4.5 Summary

This chapter has described the two empirical studies, including the methodology, approach, implementation. These two studies were designed and carried out to achieve the research aim: *to understand the discrepancies between the results of Thelin* et al. *(2003) and Dunsmore* et al.*'s (2003) studies.*

Chapter 5

Empirical Study One - Results and Analysis

This chapter describes the results and analysis of the data collected from empirical study one described in chapter 4. All statistical analysis was performed using The Statistical Package for the Social Sciences (SPSS) version 11.

5.1 The Aim

This study's aim was to compare the UCR and CBR inspection techniques. This was done by measuring the number of defects detected by individual participants using the different techniques. In this study the reading technique variable was controlled. This study was to partially answer the research question: *is there a difference between the UCR and CBR inspection techniques?*

Course	UCR	CBR
Software Engineering	5	2
Computer Science	7	4
Information Technology	1	4
Unknown	0	2

Table 5.1: Student degree course by inspection method.

5.1.1 The Study

Twenty–five students participated in this study: 12 performed the UCR inspection and 13 performed the CBR inspection. As students entered the room they were assigned the inspection technique the previous participant was not assigned. This was to randomise the allocation of each inspection technique.

Table 5.1 shows the student participants by degree course. This was collated from the demographics form completed by each participant. Two participants are shown as unknown. This is because they did not check that box on the demographic form. However, they were students from within the Department of Computing.

5.2 The Results

The following sections describe the results of Empirical Study One.

5.2.1 All Defects

Figure 5-1 depicts the comparison between the UCR and CBR code inspection techniques. It displays the total number of times each of the 18 seeded defects were discovered by the different techniques. Table 5.2 displays the total number of defects, the subset of defects due to delocalisation and the false positives generated by each technique. Table 5.3

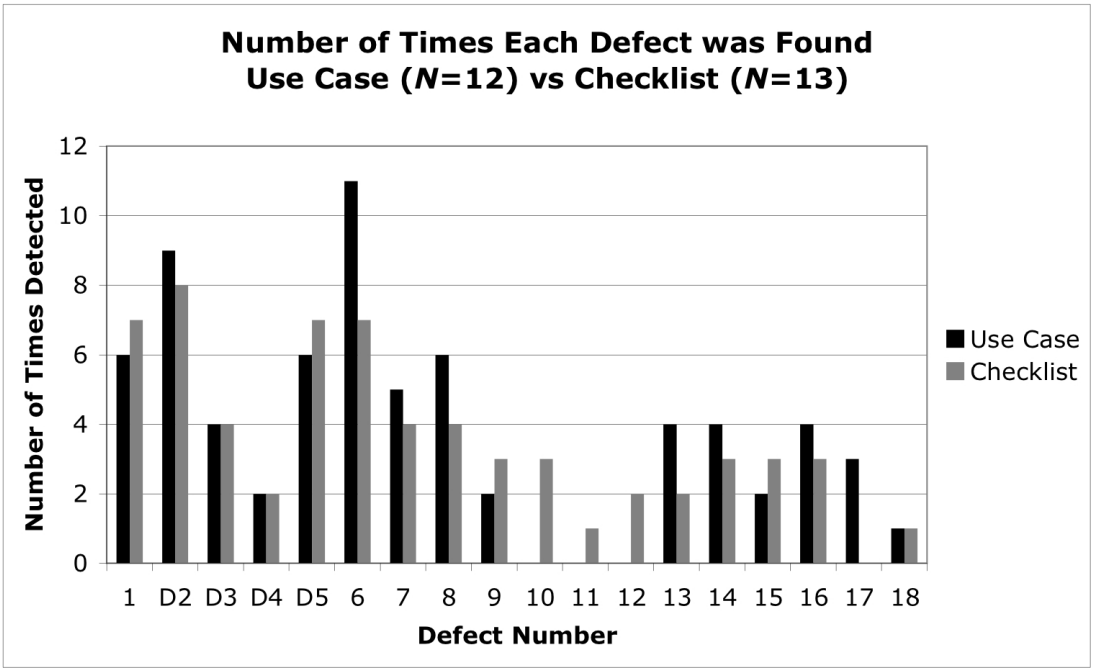

Figure 5-1: Number of times each defect was discovered by the different techniques. D2, D3, D4, D5 were due to delocalisation.

	Technique	
Defect type	**CBR (N= 13)**	**UCR (N= 12)**
All defects	4.9	5.7
Delocalised	1.6	1.7
False positives	2.2	4.1

Table 5.2: Mean number of defects discovered and false positives generated by each technique for all participants.

presents an overall summary of Empirical Study One's results. Defects, both local and delocal in nature, were included here.

Tables 5.2 and 5.3, and Figure 5-1, clearly show that the two techniques returned similar results. There is little difference between the total number of defects detected by each technique. The defect distribution detected by each method, shown in Figure 5-2, indicates CBR to be more widely spread around the median value than the UCR technique. Figure 5-2 also shows there are no suspected outliers within this data.

		Technique	
		CBR (N=12)	**UCR (N=13)**
Defects total of 18	Mean	5.3	5.5
	Std. Deviation	4.2	3.1
	Std. Error Mean	1.2	0.9
	Minimum	0	1
	Maximum	11	11
False Positives	Mean	2.4	3.8
	Std. Deviation	2.7	3.2
	Std. Error Mean	0.8	0.9
	Minimum	0	1
	Maximum	5	12

Table 5.3: Summary of study one results.

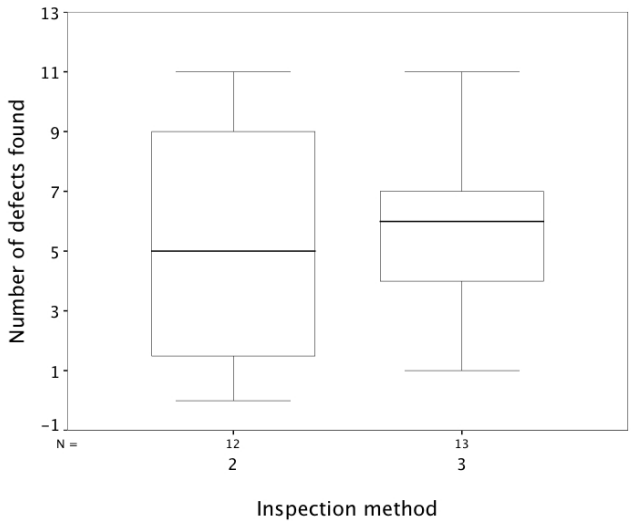

Figure 5-2: Boxplot of the defects found by each technique.

CBR vs UCR			
t	**df.**	**p**	**Sig.**
-0.1	23	0.9	$p < 0.05$

Table 5.4: Independent samples t-test results for all defects.

The collated data was analysed to determine if there was a statistically significant difference between the two inspection techniques. This was to answer the research question posed at the beginning of this chapter: *is there a significant difference between the UCR and CBR inspection techniques?*

To determine if there was a significant difference, an independent samples t-test was conducted upon the method used and the total number defects detected. Table 5.4 displays the results.

The independent samples t-test ($p = 0.9$) indicates there was no statistically significant difference between the UCR and CBR inspection techniques since $p > 0.05$. This indicates that neither method outperformed nor under performed the other in enabling or hindering defect discovery.

5.2.2 Delocalised Defects

Figure 5-3 depicts a graphical comparison between the two techniques, showing the number of times the four defects due to delocalisation were detected. Table 5.5 summarises the delocalisation defect data.

A significance test was carried out on the data for defects due to delocalisation in a similar manner to all defects (see section 5.2.1). This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the UCR and CBR inspection techniques?* Table 5.6 displays the significance test results.

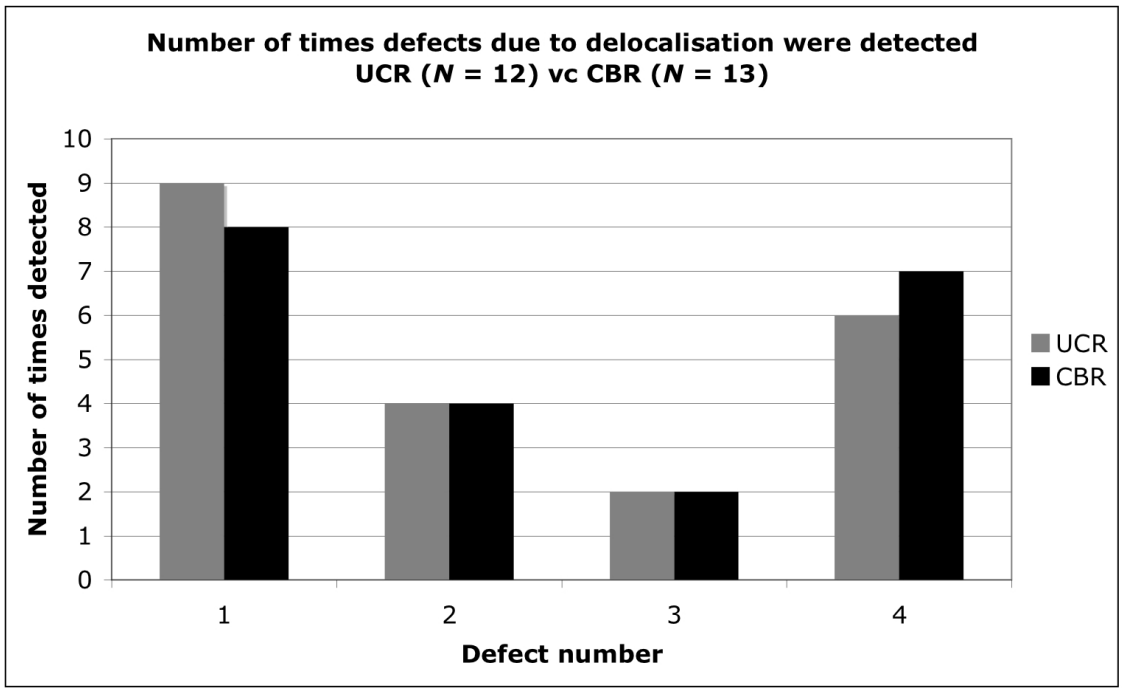

Figure 5-3: Total number of defects due to delocalisation detected by the different techniques.

		Inspection Technique	
		CBR (N=12)	**UCR (N=13)**
Defects total of 4	Mean	1.7	1.6
	Std. Deviation	1.3	1.2
	Std. Error Mean	0.4	0.3
	Minimum	0	0
	Maximum	4	3

Table 5.5: Summary of study one delocalised defect results

CBR vs UCR			
t	**df.**	**p**	**Sig.**
0.3	23	0.8	$p < 0.05$

Table 5.6: Independent samples t-test results for delocal defects.

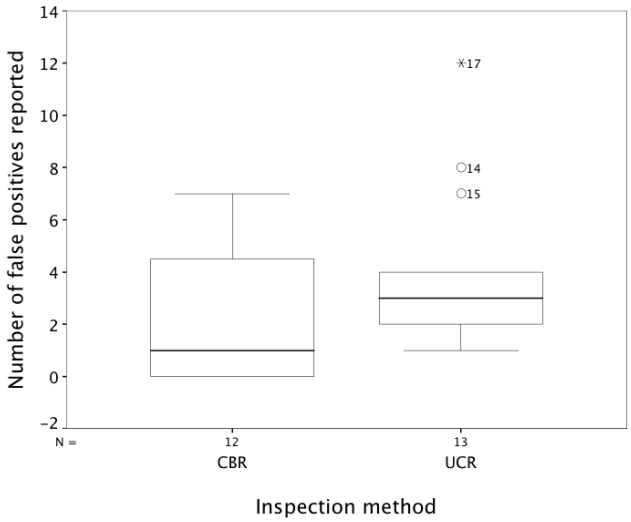

Figure 5-4: Boxplot of false positives generated by the two techniques.

The independent samples t-test ($p = 0.8$) indicates there was no statistically significant difference between the UCR and CBR inspection techniques for detecting defects due to delocalisation since $p > 0.05$. This indicates that neither method outperformed nor under performed the other in enabling or hindering the discovery of delocalised defects.

5.2.3 False Positives

The number of false positives generated by each technique was collected and collated from the data. The CBR technique had a mean of 2.4 while the UCR technique had a mean of 4.9. It is immediately obvious that there is a difference between the two techniques. The boxplots shown in Figure 5-4 indicates three possible outliers within the UCR technique.

The defect reporting forms submitted by the three possible outliers were closely examined. This was to investigate whether or not this data should be excluded from the study. The defect reporting forms from two of the outlier participants indicates they were looking for performance and optimisation issues, not defects as defined in section 4.2.1. This may be a reflection on the fact that at the time of the study these two students were enrolled in the unit *Software Engineering Tools and Metrics 352*, which looks closely at both performance

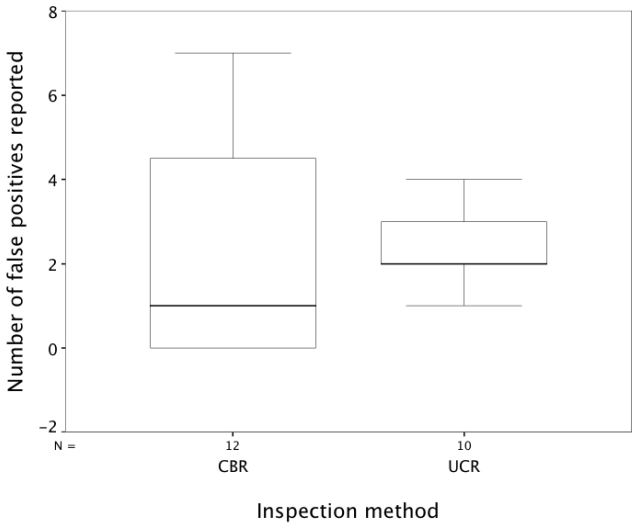

Figure 5-5: Boxplot of false positives with outliers omitted.

and optimisation. An analysis of participant 14's defect reporting form indicated that this participant had very little knowledge of either the Java^{TM} language or UML.

The first two outliers did not fulfill the requirement of the study, to search for defects within the code, as defined in section 4.2.1. On this basis their data was excluded from the false positives' results.

On the basis of participant 14's defect reporting form analysis, this participants details were also excluded from the results. Figure 5-5 is the boxplot with the outliers omitted. With the outliers removed the CBR technique had a mean of 2.4 and the UCR technique had a mean of 2.3.

A significance test was carried out on the false positives data from this study in a similar manner to that carried out for all defects (see section **??**). This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the UCR and CBR inspection techniques?* Table 5.7 displays the results.

The independent samples t-test ($p = 0.9$) indicates there was no statistically significant difference between the UCR and CBR inspection techniques for generating false positives

CBR vs UCR			
t	**df.**	**p**	**Sig.**
0.1	14.1	0.9	$p < 0.05$

Table 5.7: Independent samples t-test results for the number of false positives generated.

since $p > 0.05$. This indicates that neither method performed better or worse than the other in generating false positives.

5.3 Summary

This chapter outlined the aims of Empirical Study One and the independent variable being controlled. It then showed the overall number of defects detected by each technique and the results of the statistical analysis performed using them. It then listed and statistically analysed the number of defects due to delocalisation detected as well as the false positives generated. The researcher found no statistically significant difference between the UCR and CBR inspection techniques.

Chapter 6

Empirical Study Two - Results and Analysis

This chapter describes the results and analysis of the data collected from Empirical Study Two described in Chapter 4.

This chapter is divided into three sections. The first section examines and analyses the student grouping results. The second section examines and analyses the industry professional grouping results. The third section compares the student and industry professional grouping results.

6.1 The Aim

This study's aim was to compare the UCR, CBR and UBR inspection techniques. This was done by measuring the number of defects detected by the individual participants using the different techniques. In this study the participant's experience level variable was controlled by two separate participant groupings. The first group were students. The second group were industry professionals.

Course	UCR	CBR	UBR
Software Engineering	1	11	9
Computer Science	8	3	0
Information Technology	1	0	3

Table 6.1: Student degree course by inspection method.

There were two research questions to be partially answered by Study Two. The first one: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* The second one: *is there a significant difference between the number of defects detected by participants with different experience levels?*

By controlling the experience level and reading technique variables, Study Two supported the research aim *to understand the discrepancies between the results reported by Thelin* et al. *(2003) and Dunsmore* et al. *(2003).*

6.2 The Students

This section examines the results reported by the student participants.

Thirty–six students participated in this study: 10 performed the UCR inspection, 12 performed the UBR inspection and 14 performed the CBR inspection. Due to the unknown number of participants prior to commencement, the first 26 students were assigned either the UBR or CBR technique. As the UCR technique was tested in study one the researcher decided this would not be tested again unless there were sufficient participant numbers. The high participant turn out permitted the final 10 students to be assigned the UCR technique.

Table 6.1 shows the students by degree course. This was collated from the demographics form completed by each participant.

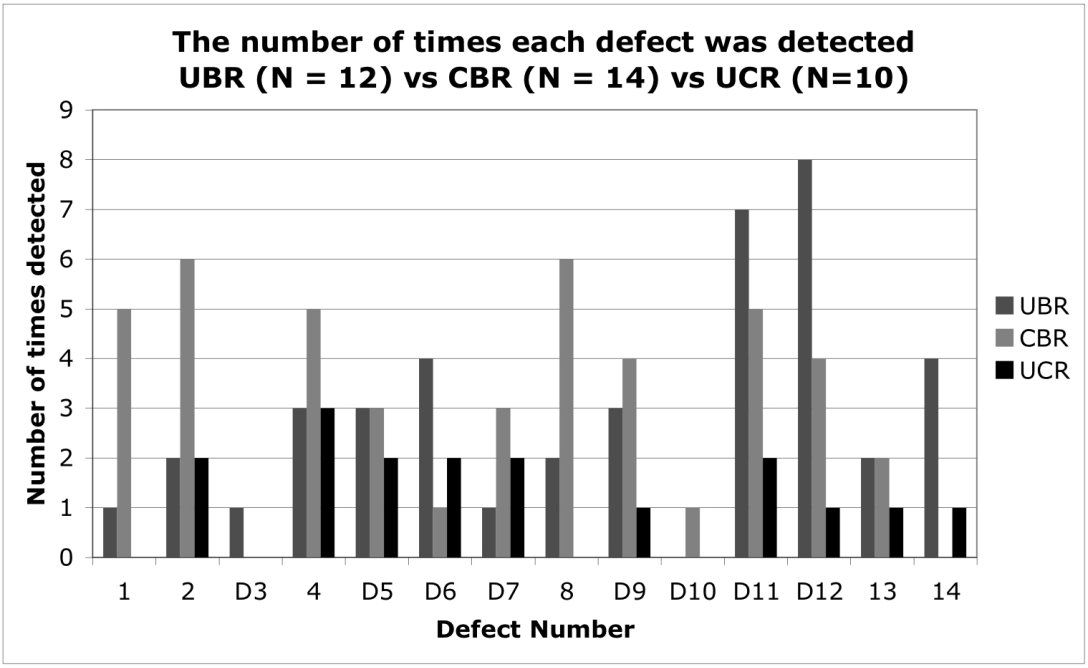

Figure 6-1: Number of times each defect was discovered by the different techniques. Defects labelled with a D were due to delocalisation.

6.2.1 All Defects

Figure 6-1 depicts the total number of times each of the 14 seeded defects were discovered by the different techniques. Table 6.2 presents an overall summary of Study Two's results. Defects, both local and delocal in nature were included within these representations.

Table 6.2 and Figure 6-1, show the CBR and UBR techniques returned similar results. There is little difference between the total number of defects detected by those two techniques. The UCR technique returned a lower number of defects than the CBR and UCR but also generated the least number of false positives.

The median detection values were close together, as can be seen in Figure 6-2, which also shows a possible outlier within the CBR data. A close examination of participant 21's defect reporting form indicated that this participant took 78 minutes for the inspection. This was 24 minutes longer any other student using the same technique. This participant reported no false positives. From examining the defect reporting form no abnormalities

		Inspection Technique		
		CBR (N= 14)	**UCR (N= 10)**	**UBR (N= 12)**
Defects total of 14	Mean	3.2	1.7	3.4
	Std. Deviation	2.5	1.7	2.7
	Std. Error Mean	0.7	0.5	0.8
	Minimum	0	0	0
	Maximum	10	5	9
False Positives	Mean	2.0	0.8	0.8
	Std. Deviation	1.9	1.9	1.7
	Std. Error Mean	0.5	0.6	0.5
	Minimum	0	0	0
	Maximum	6	6	6

Table 6.2: Summary of study two results for students.

UBR vs CBR vs UCR			
df.	**F**	**Sig.**	**Significant if:**
2	1.7	0.2	Sig. < 0.05

Table 6.3: One–way ANOVA results for all defects.

were seen questioning its validity. Therefore this participant's results remained within the data set.

The collated data was analysed to determine if there was a statistically significant difference between the three inspection techniques. This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* A one–way ANOVA test was performed because there were more than two population samples to be tested. It was performed on the method used and the number of defects detected by each method. Table 6.3 displays the results. The one–way ANOVA results indicate no statistically significant difference between the UBR, CBR and UCR inspection techniques since Sig. > 0.05. These three inspection techniques did not outperform nor under perform in enabling or hindering the discovery of defects.

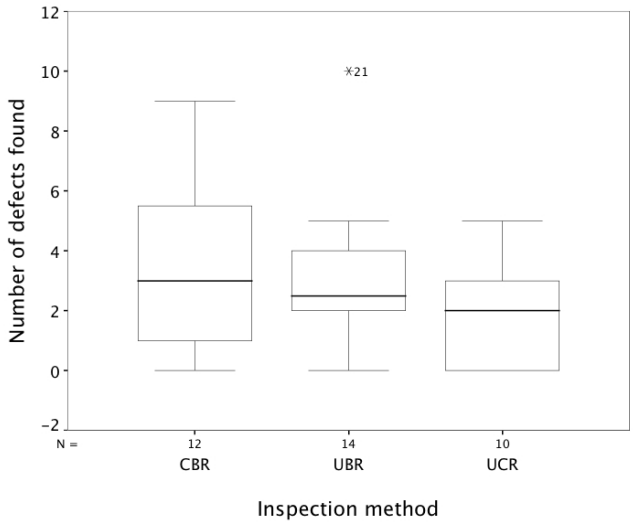

Figure 6-2: Boxplots of the defects found by the three techniques.

		Inspection Technique		
		CBR (N= 14)	**UCR (N= 10)**	**UBR (N= 12)**
Defects (total of 8)	Mean	1.5	1.0	2.2
	Std. Deviation	1.4	1.1	1.9
	Std. Error	0.4	0.3	0.5
	Minimum	0	0	0
	Maximum	5	5	3

Table 6.4: Summary of study two delocalised defect results for student participants.

6.2.2 Delocalised Defects

Figure 6-3 depicts a graphical comparison between the three techniques, showing the number of times the eight defects due to delocalisation were detected. Table 6.4 summarises the delocalisation defect data.

The data for delocalised defects was then analysed to determine if there was a statistically significant difference between the three inspection techniques. This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* A one–way ANOVA test was performed upon the method used and the number of defects due to delocalisation that

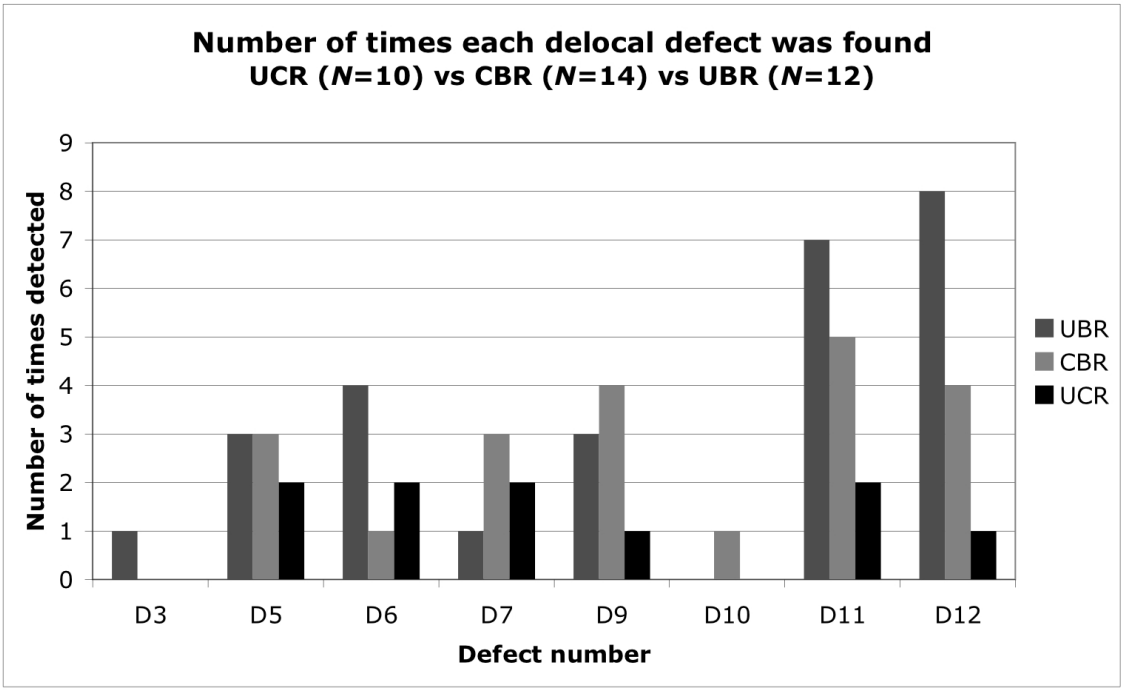

Figure 6-3: Total number of defects due to delcoalisation detected by the different techniques.

UBR vs CBR vs UCR			
df.	**F**	**Sig.**	**Significant if:**
2	1.9	0.2	Sig. < 0.05

Table 6.5: One–way ANOVA results for delocal defects.

were detected. Table 6.5 displays the results.

The one–way ANOVA test indicate no statistically significant difference between the UBR, CBR and UCR inspection techniques for detecting defects due to delocalisation since Sig. > 0.05. The three inspection techniques did not outperform nor under perform in enabling or hindering the detection of defects due to delocalisation.

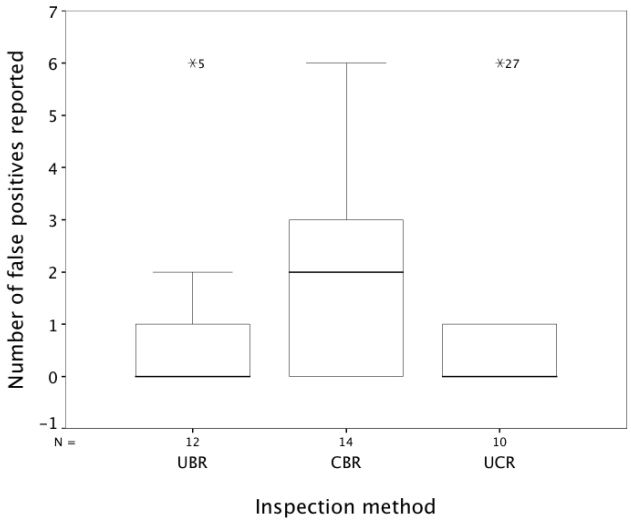

Figure 6-4: Boxplots of false positives generated by the three techniques.

6.2.3 False Positives

Figure 6-4 displays the number of false positives generated by the three techniques. There is an obvious difference between the three techniques. The boxplots shown in Figure 6-4 indicate two possible outliers, one within the CBR technique and, one within the UCR technique.

Examining the defect reporting form belonging to participant five (the possible outlier in the UBR sample) indicates that when the code under inspection referred to external classes, the participant did not understand this. Therefore these were recorded as defects. The false positives generated by this participants were in fact delocalised references. This participant's reporting form highlights the problems delocalisation causes within OO code inspections. This participant's results remained within the data set.

Examining the defect reporting form belonging to participant 27 (the possible outlier in the UCR sample) indicates this participant did not understand the code. Therefore false positives were reported. The participant appears to have understood the task they were to perform and had the required level of knowledge of Java^{TM} and UML. This participant's results also remained within the data set.

UBR vs CBR vs UCR			
df.	**F**	**Sig.**	**Significant if:**
2	1.8	0.2	Sig. < 0.05

Table 6.6: One–way ANOVA results for delocal defects.

A significance test was carried out on the false positives data. This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* A one–way ANOVA test was performed. Table 6.6 displays the results.

The one–way ANOVA result indicates no statistically significant difference between the UBR, CBR and UCR inspection techniques for generating false positives since Sig. $>$ 0.05. As with the detection of all defects (section 6.2.1) and the detection of defects due to delocalisation (section 6.2.2) this indicates that these three inspection techniques did not outperform nor under perform in generating false positives.

6.3 Industry Professionals

This section examines the results reported by the industry professional participants.

Eighteen industry professionals participated in this study: 10 performed the UCR inspection and 8 performed the CBR inspection. Due to the small number of available industry participants the researcher decided to compare the CBR and UCR inspection techniques within this grouping. There were two reasons behind this decision. First, as described in the Background section, CBR is the standard inspection technique used within industry. Second, in Thelin *et al.*'s (2003) study, a UBR inspection was conducted with participants having considerable industry experience. Therefore the researcher chose the UCR technique, as this had not previously been reported upon.

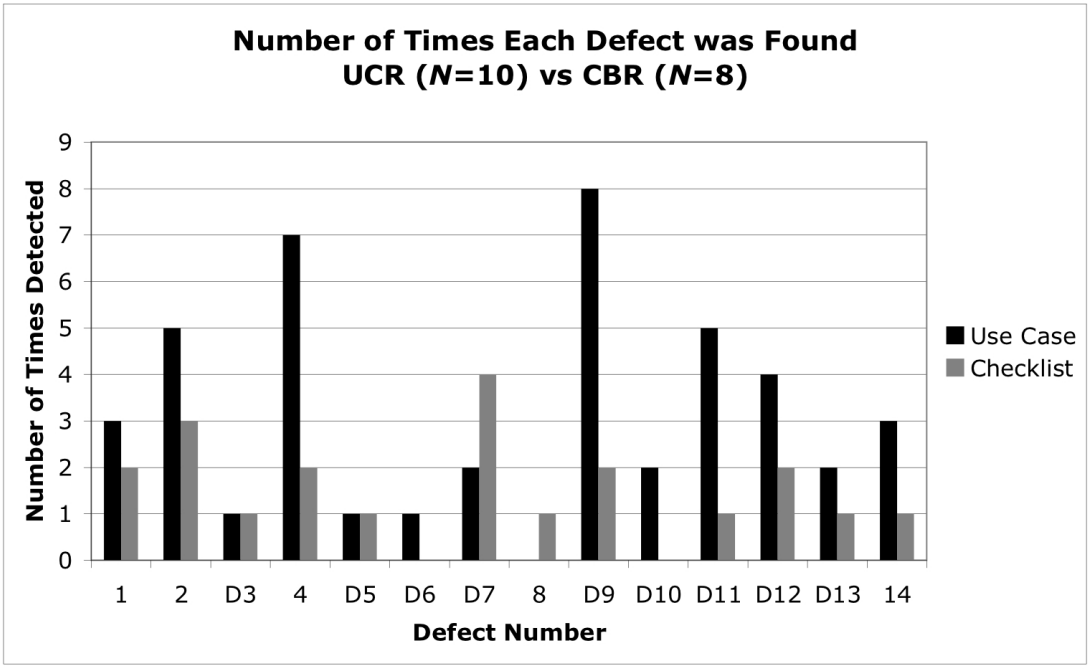

Figure 6-5: Industry: Number of times each defect was discovered by the different techniques. Defects labelled with a D were due to delocalisation.

	Technique	
Defect type	**CBR (N= 8)**	**UCR (N= 10)**
All defects	5.6	4.4
Delocalised	1.4	2.4
False positives	0.5	0.0

Table 6.7: Industry: mean defects detected and false positives generated by each technique.

6.3.1 All Defects

Figure 6-5 depicts the graphical comparison between the UCR and CBR techniques. It displays the total number of times each technique detected the 14 seeded defects. Table 6.7 displays the mean number of defects, the subset of defects due to delocalisation and the false positives generated by each technique. Table 6.8 presents an overall summary of the results from Study Two. Defects, both local and delocal in nature are included within these representations.

Tables 6.7 and 6.8, and Figure 6-5, clearly show the two techniques returned similar

		Technique	
		CBR (N= 8)	**UCR (N= 10)**
Defects total of 14	Mean	5.6	4.4
	Std. Deviation	1.9	2.0
	Std. Error Mean	0.7	0.6
	Minimum	0	1
	Maximum	11	11
False Positives	Mean	0.5	0.0
	Std. Deviation	0.8	0.0
	Std. Error Mean	0.3	0.0
	Minimum	0	0
	Maximum	2	0

Table 6.8: Summary of study two industry results.

results for the inspection. There is little difference between the total number of defects detected by either technique. The boxplot shown in Figure 6-6 indicates a possible outlier in the CBR sample. Examining participant two's defect reporting form does not indicate a misunderstanding of the task nor that their knowledge of Java™ or UML was insufficient. From this inspection, there is no justifiable reason for this participant's data to be removed from the sample. Therefore it remains in the data set.

The data were analysed to determine if there was a statistically significant difference between the two inspection techniques for industry professionals. This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* For industry professionals, this research question must be modified because only CBR and UCR were able to be tested.

The nonparametric nature of this data, as seen in Figure 6-6, meant that a Mann-Whitney test was carried out upon the total number of defects returned, and the inspection technique used to determine whether or not there was a significant difference between the techniques.

The Mann-Whitney result (Sig. = 0.1) indicates no statistically significant difference

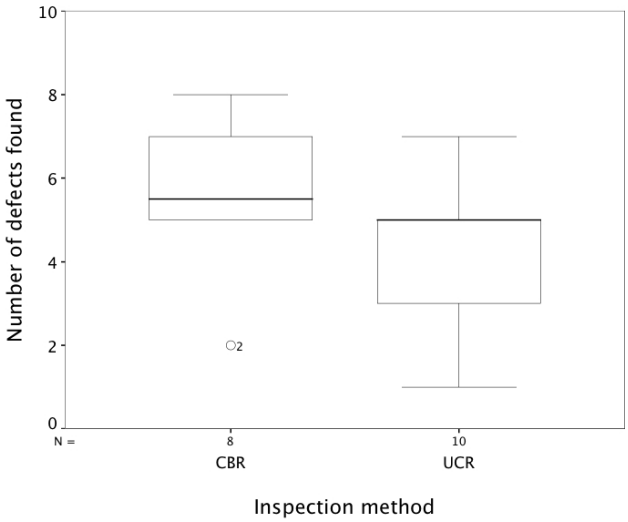

Figure 6-6: Boxplot of defects found by the two techniques (industry).

		Inspection Technique	
		CBR (N= 8)	**UCR (N= 10)**
Defects total of 8	Mean	2.6	2.4
	Std. Deviation	1.1	1.4
	Std. Error Mean	0.4	0.5
	Minimum	1	1
	Maximum	4	5

Table 6.9: Summary of study two, delocalised defect results (industry).

between the UCR and CBR inspection techniques when applied by industry professionals, since Sig. > 0.05. This indicates that neither technique outperformed nor under performed in enabling or hindering the discovery of defects.

6.3.2 Delocalised Defects

Figure 6-7 depicts a graphical comparison between the two techniques, specifically showing the number of times the eight defects due to delocalisation were detected. Table 6.9 summarises the delocalisation defect data.

A Mann-Whitney test was carried out on the defects due to delocalisation data in a

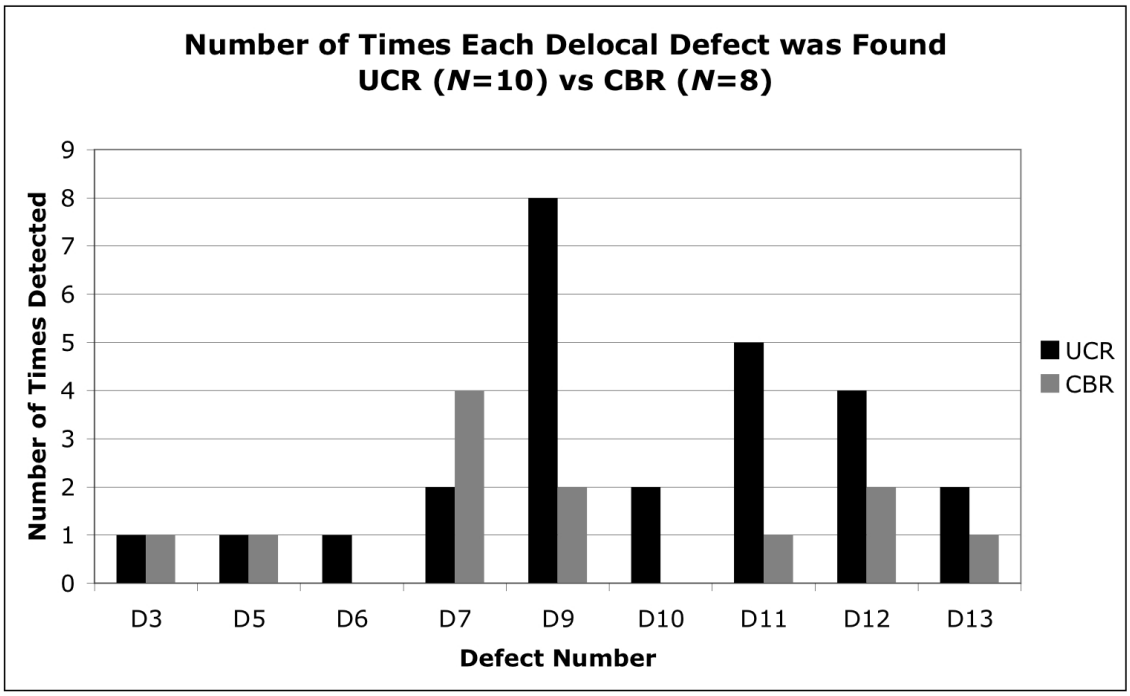

Figure 6-7: Total number of times each delocal defect found by the different techniques.

similar manner as for all defects (see section 6.3.1). This was to partially answer the research question posed at the beginning of this chapter: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* For industry professionals, this research question must be modified as only CBR and UCR were able to be tested.

The Mann-Whitney result (Sig. = 0.6) indicates no statistically significant difference between the UCR and CBR inspection techniques for detecting defects due to delocalisation when applied by industry professionals since $p > 0.05$. This indicates that neither method outperformed nor under performed in enabling or hindering the discovery of delocalised defects.

6.3.3 False Positives

The number of false positives generated by each technique were collected and collated from the data. The CBR technique generated 4 while the UCR technique did not generate any.

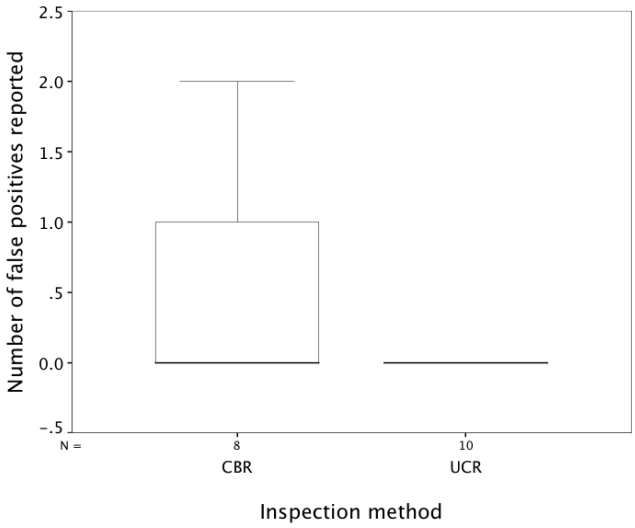

Figure 6-8: Boxplots of false positives generated by the two techniques (industry).

CBR vs UCR			
t	**df.**	**p**	**Sig.**
1.9	7.0	0.1	$p < 0.05$

Table 6.10: Independent samples t-test results for false positives (industry).

It is clear that there is very little difference between the two techniques. The box plot shown in Figure 6-8 indicates no outliers. The boxplot's unusual look is due to the fact that the UCR method did not generate any false positives.

A significance test was carried out on the false positives data from this study in a similar manner to that carried out for all defects (see section 6.3.1). This was to partially answer the research question given at the beginning of this chapter: *is there a significant difference between the UCR, CBR and UBR inspection techniques?* For the industry participants grouping this question was modified, removing the UBR inspection technique.

Table 6.10 displays the results. The independent samples t-test result ($p = 0.1$) indicates no statistically significant difference between the UCR and CBR inspection techniques applied by industry professionals in generating false positives since $p > 0.05$. This indicates that neither method performed better or worse in generating false positives.

	Technique		
Defect type			
Students	**UCR (N= 10)**	**CBR (N= 14)**	**UBR (N= 12)**
All defects	1.7	3.2	3.4
Delocal	1.0	1.5	2.2
False positives	0.8	2.0	0.8
Industry	**N= 10**	**N= 8**	NA
All defects	4.4	5.6	NA
Delocal	2.5	1.4	NA
False positives	0.0	0.5	NA

Table 6.11: Mean of defects results summary: Student vs Industry.

6.4 Students and Industry Professionals

One aim from empirical Study Two was to answer the research question: *is there a significant difference between the number defects detected by participants with different experience levels?* To achieve this the participant experience level variable was controlled. This section compares the overall results returned by the student grouping with the results returned by the industry professional grouping.

Table 6.11 breaks down the data from the two groupings. Ten student participants performed the UCR inspection returning a summed total of 17 defects while eight industry participants, using the same technique return a summed total of 44 defects.

Table 6.2 indicates the student participants discovered a mean of 1.7 and 3.2 defects using the UCR and CBR techniques respectively. Table 6.8 indicates that for the same techniques, industry participants discovered a mean of 5.6 and 4.4 defects respectively.

Another difference between the two groups is seen in the false positives generated by the different grouping using the CBR technique. This can be seen in Table 6.11. Student participants using the CBR generated a total of 28 false positives, while industry participants generated only 4 false positives.

CBR vs UCR			
t	**df.**	**p**	**Sig.**
-3.2	52	0.003	$p < 0.05$

Table 6.12: Independent samples t-test: Student and Industry groupings for the total number of defects detected.

CBR vs UCR			
t	**df.**	**p**	**Sig.**
-2.1	52	0.04	$p < 0.05$

Table 6.13: Independent samples t-test: Student and Industry groupings for the total number of defects due to delocalisation detected.

As has already been clearly stated earlier in this study, one aim from empirical Study Two was to answer the research question: *is there a significant difference between the number defects detected by participants with different experience levels?* To achieve this the participant experience level variable was controlled.

Independent samples t-tests were conducted on the total number of defects discovered, the total number of defects due to delocalisation and the total number of false positives generated between the two groupings.

Table 6.12 displays the results for the comparison between total number of defects discovered. Table 6.13 displays the results for the defects due to delocalisation and Table 6.14 displays the false positives results.

The data in Tables 6.12, 6.13 and 6.14 demonstrate a statistically significant difference on

CBR vs UCR			
t	**df.**	**p**	**Sig.**
3.1	45.3	0.003	$p < 0.05$

Table 6.14: Independent samples t-test: Student and Industry groupings for the total number of false positives generated.

all three accounts.

The data collected and results presented from Study Two clearly answer the research question posed at the beginning of the chapter: *is there a significant difference between the number defects detected by participants with different experience levels?* Yes, there is a statistically significant difference between these two groupings. Industry professionals detected a statistically significant higher number of defects and generated a statistically significant lower number of false positives than the student participants.

Chapter 7

Discussion and Conclusions

This chapter first discusses the statistical analysis results reported in Chapter 5 and Chapter 6. It then examines the confounding factors that influenced this research. It examines these results in light of the overall aims stated in Chapter 1. It also examines these results in relation to Sjoberg *et al.*'s (2005) article, *A Survey of Controlled Experiments in Software Engineering*. Finally, recommendations are made regarding the meaning of the results and the possible directions of continuing research based on this study.

7.1 Discussion

Empirical Study One compared the CBR and UCR inspection techniques with student participants. It compared these techniques first for all defects, then for defects due to delocalisation and finally for the generation of false positives. The goal of Empirical Study One was to answer the research question: *is there a difference between the UCR and CBR inspection techniques?* The results showed there to be no statistically significant difference in the detecting of non-delocalised defects, defects due to delocalisation, nor in false positive generation between the two inspection techniques. The statistical evidence from this study therefore, does not support the results of Dunsmore *et al.* (2003) that UCR performed worse than CBR.

Empirical Study Two first compared the CBR, UBR and UCR inspection techniques with student participants. As in Empirical Study One, it compared the techniques first for all defects, then for defects due to delocalisation and finally for the generation of false positives. A goal of Empirical Study Two was to answer the research question: *is there a significant difference between the CBR, UCR and UBR inspection techniques?* This question was answered as the results showed there to be no statistically significant difference in the detection of non-delocalised defects, defects due to delocalisation, nor in false positive generation between the UCR, CBR and UBR inspection techniques with student participants. For student participation, the results do not support the results of Dunsmore *et al.* (2003) stating UCR performed worse than CBR. The researcher is also unable to agree with Thelin *et al.* (2003) stating UBR performed more effectively than CBR.

The second part to Empirical Study Two compared the CBR and UCR inspection techniques with industry professional participants. A goal of this section from Empirical Study Two was to use the industry participants to answer the research question: *is there a difference between the UCR and CBR inspection techniques?* As with student participants, the study showed there to be no statistically significant difference in the detection of defects due to delocalisation, nor in false positive generation, between the UCR and CBR inspection techniques with industry participants. These results do not support the results of Dunsmore *et al.* (2003) stating UCR performed worse than CBR.

Empirical Study Two therefore answered the two research questions it posed:

1. Is there a significant difference between the CBR, UCR and UBR inspection techniques?
2. Is there a significant difference between the number defects detected by participants with different experience levels?

There was no statistically significant difference for the detection of non-delocalised defects, defects due to delocalisation nor the number of false positives generated by the UCR,

CBR and UBR inspection techniques when utilised by the student grouping. In the same manner, there was no statistically significant difference between the CBR and UCR techniques when utilised by the industry professional grouping.

However, there ***was*** a statistically significant difference for the detection of non-delocalised defects, defects due to delocalisation and the number of false positives generated between the two independent groupings, students and industry professionals.

7.2 Confounding Factors

In carrying out this empirical research there were two types of influences, or threats, that may have affected the results: threats to the internal validity, and threats to the external validity of the study. This section examines the threats to the results that have arisen from these two empirical studies.

7.2.1 Internal Validity

Selection effect occurs when there is a difference between the ability of the participants, which is not evenly spread throughout the study. Information related to the academic performance of student participants was not available for this study. All student participants were final year or honours level students meeting the criterion that they had passed *Software Engineering 252/552* or equivalent.

A participant's lack of enthusiasm could affect the results. As this study was voluntary, those who gave up personal time to partake, with no direct benefits for themselves, do not suffer from a lack of enthusiasm. The over–enthusiasm of participants may, however, affect the results of the study. It is possible that only the highly ambitious and self–motivated students took part in these studies.

The second threat was that of a steep learning curve. Although the theory of code inspections is taught within one unit at Curtin University's Department of Computing, this unit does not give students practical experience in this area. All students were introduced to the practical application of the inspection technique they used when they participated in the study. This was a new skill participants were acquiring, the learning curve required may have been steep and therefore prevented participants from discovering a higher number of defects.

7.2.2 External Validity

1. In empirical study one, all participants were students at similar stages within their undergraduate degrees. This population may not represent the wider software engineer community. For this reason, 33% of participants within study two were industry professionals.
2. The difference in results between the UCR and the CBR and UBR techniques in empirical study two for student participants, may be related to the demographic breakdown (seen in Table 6.1). The vast majority of these participants are Computer Science students whereas the CBR and UBR have a majority of Software Engineering students as participants.
3. The defects within the code were artificially seeded and may not reflect those that are currently experienced within the larger software engineering environment. Catering for this is explained in section 4.2.3.
4. This study focused upon the individual role within the different phases of a typical inspection. This may not be representative of normal industry practice and may have impacted the inspectors performance. It did provide the ability to examine the results of the inspection techniques as used by individual inspectors.
5. The sample size of industry professionals was 18. In context, this was 22.8% of the entire sample size (study one and two) and made up 33% of study two participants.
6. This small number of industry professionals did not permit all three inspection techniques (UCR, CBR and UBR) to be compared within the context of this study.

The results and recommendations of this research need to be considered with the above listed validity threats kept in mind. The study should be repeated with a larger industry professional sample size in order to compare the three techniques UBR, CBR and UCR with each other.

7.3 Aims

7.3.1 Aim 1

Understanding the discrepancies between the results of Dunsmore et al. *(2003) and Thelin* et al. *(2003)* was fulfilled by Empirical Study One and Two. Empirical Study One compared the UCR and CBR inspection techniques while controlling the participant experience level variable. All participants were students that had successfully completed *Software Engineering 252/552*.

Empirical Study Two continued to fulfil this aim by establishing two participant groupings. Students formed the first grouping while industry professionals formed the second grouping. This study allowed for the reading technique and experience level variables to be controlled.

Empirical Study One showed no statistically significant difference between the CBR and UCR inspection techniques. This disagrees with the reported results that the UCR technique was the least effective (Dunsmore *et al.*, 2003, p. 685)

Empirical Study Two also showed no statistically significant difference between the UBR, CBR and UCR inspection techniques within the student participant grouping. There was also no statistically significant difference between the CBR and UCR inspection techniques within the industry professional grouping. These results disagree with the reported results that the UCR technique's performance was the least effective, and it also disagrees with

the reported results that the UBR was more effective than CBR in detecting defects (Dunsmore *et al.*, 2003; Thelin *et al.*, 2003).

The data analysis carried out in empirical Study Two, comparing the student and industry professional groupings did reveal a statistically significant difference between the two participant groupings. The industry professional grouping had statistically significantly higher results for detecting both localised and delocalised defects, while generating fewer false positives than the student grouping.

Therefore the discrepancies between Thelin *et al.* (2003) and Dunsmore *et al.*'s (2003) studies results might be explained by differences in the participants' experience levels.

7.3.2 Aim 2

Comparing the OO code inspection techniques: UCR, CBR and UBR to determine the technique that detects the highest number of defects within a given period of time, was fulfilled by empirical Study One and Two. In both studies the independent variables, the reading technique and the participants' experience levels were controlled. The first study, comparing UCR and CBR demonstrated no statistically significant difference between the methods within the 120-minute time period. For the student participants grouping in study two there was no significant difference between the UCR, CBR and UBR techniques. This same result was obtained for the industry professional grouping within this study.

Therefore, there is no statistical difference between the three inspection techniques CBR, UBR and UCR in the number of defects detected or false positives generated in a given time period, for these studies in 120 minutes.

7.3.3 Aim 3

Section 7.5 reports on *making recommendations regarding effective OO code inspection techniques.*

7.4 Software Engineering Empirical Studies

Research into empirical studies performed in the software engineering discipline indicated concerns that the vast majority of participants are students (87%), with only a small number (9%) being industry professionals (Sjoberg *et al.*, 2005, p. 738). The use of both student and industry professionals within this study has provided objective data that tests the validity of this concern. The data collected and analysed has clearly demonstrated a statistically significant difference between the student and professional participants. This difference must be considered before generalising the results.

7.5 Recommendations

From this research there are three recommendations:

1. Empirical studies in software engineering performed using student participants cannot be generalised to the wider software engineering industry. The difference between the two groups is significant enough to warrant this should not happen. Empirical research that has used student participants should only be generalised, if possible, to the wider student body within a similar demographics setting, i.e. background, age, experience.

   Research that has reported its findings for student participants, needs to be repeated using industry participants. The reported results are only valid for a student

grouping with a similar set of demographics. This research has shown that the use of industry participants results in very different data being generated than when student participants are used.

2. Inspection techniques should be adapted to suit the inspector's experience level. Current practice is a 'one-size-fits-all' approach. Adapting the techniques used may result in a higher number of defects being detected. This would lead to better quality software being developed.
3. Inspections must be continuously developed based on an organisation's historical data. This research showed that the more experienced participants, those from industry, detected more defects than the student participants. This was accredited to the differing experience levels. Using historical data to continuously update inspections capitalises upon the organisation's experience. The more historical information an organisation has on its past successes and failures, the more information inspectors have to use as a base for the inspections. In this way the organisation's experience can be used by inspections just as an inspector relies and uses their own experiences.

7.6 Future Work

This research has laid the groundwork for many different issues to be pursued. Two of the more important areas are described here.

The first research area is in examining the adaptive application of all software development practices and processes, including inspection techniques, to assist in shortening the learning curve of these practices and processes. Experienced software developers have developed the application of these process and practices to the building of software. Can inspection techniques also be applied in an adaptive manner to bring about a shortened learning curve of these techniques? The long term goal: the production of higher quality software.

The second research area highlighted by this research is the difference between students and industry professionals. This research provides a basis for investigating the application of inspection techniques within an educational capacity. How could the application of these techniques aid in developing the skill and ability of students to read and understand code written by others?

7.7 Concluding Comment

Inspection techniques are 'no silver bullet' (Brooks, 1987). Applied in context with other software development processes and practices they can aid software developers in moving closer to the 'Holy Grail': defect free software.

Bibliography

Ackerman, A. F., Buchwald, L. S., and Lewski, F. H. (1989). Software inspections: an effective verification process. *IEEE Software*, **6**(3), 31–36.

Basili, V. R., Green, S., Laitenberger, O., Shull, F., Sorumgard, S., and Zelkowitz, M. V. (1995). The empirical investigation of perspective-based reading. Technical report, University of Maryland at College Park, College Park, MD, USA.

Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering*, **12**(2), 211–221.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison–Wesley, Massachusetts.

Brooks, F. P. (1987). No silver bullet: essence and accidents of software engineering. *IEEE Computer*, **20**(4), 10–19.

Brykczynski, B. (1999). A survey of software inspection checklists. *SIGSOFT Software Engineering Notes*, **24**(1), 82–89.

Cockburn, A. (2000). *Writing Effective Use Cases*. Addison–Wesley, Massachusetts.

Curtin University of Technology (2003). Java coding standard. Web page. *https://www.computing.edu.au/documents/JavaCodingStandard.pdf* Last accessed on the 3rd October 2005.

Curtin University of Technology (2005). Human research ethics committee. Web page. *http://research.curtin.edu.au/ethics/hrec.html* Last accessed on the 23rd October 2005.

Denger, C., Ciolkowski, M., and Lanubile, F. (2004). Does active guidance improve software inspections? a preliminary empirical study. In *IASTED International Conference Software Engineering*, pages 408–413, Innsbruck, Austria.

Dunsmore, A. (2000a). An empirical investigation of a systematic object-oriented inspection technique. Technical Report EFoCS-37-2000, Department of Computer Science, University of Strathclyde.

Dunsmore, A. (2000b). Survey of object-oriented defect detection approaches and experiences in industry. Technical Report EFoCS-36-2000, Department of Computer Science, University of Strathclyde.

Dunsmore, A. (2002). *Investigating effective inspection of object-oriented code.* Ph.D. thesis, University of Strathclyde, Glasgow.

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

Dunsmore, A., Roper, M., and Wood, M. (2000b). The role of comprehension in software inspection. *The Journal of Systems and Software*, **52**(2–3), 121–129.

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

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

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

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

Fagan, M. E. (1999). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, **2**(38), 258–287.

Gilb, T. and Graham, D. (1993). *Software Inspection*. Addison–Wesley, Wokingham.

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

Laitenberger, O. and Atkinson, C. (1999). Generalizing perspective-based inspection to handle object-oriented development artifacts. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 494–503, Los Angeles, California, United States.

Laitenberger, O. and DeBaud, J. (1997). Perspective-based reading of code documents at robert bosch gmbh. *Information and Software Technology*, **39**(11), 781–791.

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

Laitenberger, O., El-Emam, E., and Harbich, T. G. (2001). An internally replicated quasi-experiment comparison of checklist and perspective-based reading of code documents. *IEEE Transactions on Software Engineering*, **27**(5), 387–421.

Leedy, P. D. and Ormrod, J. E. (2005). *Practical Research*. Pearson Merrill Prentice Hall, eighth edition.

Lejter, M., Meyers, S., and Reiss, S. P. (1992). Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, **18**(12), 1045–1052.

Martin, J. and Tsai, W. T. (1990). N-fold inspection: a requirements analysis technique. *Communications of the ACM*, **33**(2), 225–232.

Meyer, B. (1986). Genericity versus inheritance. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 391–405, Portland, Oregon, United States.

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

National Institute of Standards and Technology (2003). The economic impacts of inadequate infrastructure for software testing: final report. RTI Health, Social, and Economics Research.

Olofsson, M. and Wennberg, M. (1996). Statistical usage inspection. Master's thesis, Department. of Communication Systems, Lund Institute of Technology and Ericsson Telecom AB.

Parnas, D. L. and Weiss, D. M. (1985). Active design reviews: principles and practices. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 132–136, London, England.

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

Porter, A. A. and Votta, L. G. (1994). An experiment to assess different defect detection methods for software requirements inspections. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 103–112, Sorrento, Italy.

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

President's Information Technology Advisory Committee (1999). Information technology research: Investing in our future. Technical report, National Coordination Office for Computing, Information, and Communications, Washington, D.C. www.ccic.gov/ac/report.

Robson, C. (2002). *Real World Research*. Blackwell, Massachusetts, second edition.

Schneider, G. M., Martin, J., and Tsai, W. T. (1992). An experimental study of fault detection in user requirements documents. *ACM Transactions on Software Engineering Methodology*, **1**(2), 188–204.

Shull, F., Basili, V., and Rus, I. (2000). How perspective-based reading can improve requirements inspections. *Computer*, **33**(7), 73–79.

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

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

Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM*, **31**(11), 1259–1267.

Sommerville, I. (2001). *Software Engineering.* Addison–Wesley, Harlow, second edition.

Standards Coordinating Committee of the Computer Society of the IEEE (1983). *ANSI/IEEE Std. 729-1983 Standard Glossary of Software Engineering Terminology.* IEEE Standards Board.

Thelin, T., Petersson, H., and Wohlin, C. (2001a). Sample–driven inspection. In *WISE'01: Proceedings of the 1st Workshop on Inspection in Software Engineering*, pages 81–91, Paris France.

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

Thelin, T., Runeson, P., Wohlin, C., Olsson, T., and Andersson, C. (2002). How much information is needed for usage-based reading? A series of experiments. In *ISESE '02: Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)*, pages 127–138, Nara, Japan.

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

Travassos, G., Shull, F., Fredericks, M., and Basili, V. R. (1999). Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented pro-*

gramming, systems, languages, and applications, pages 47–56, Denver, Colorado, United States.

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

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

Wilde, N. and Huitt, R. (1992). Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, **18**(12), 1038–1044.

Wilde, N., Matthews, P., and Huitt, R. (1993). Maintaining object-oriented software. *IEEE Software*, **10**(1), 75–80.