INVESTIGATING EFFECTIVE INSPECTION OF OBJECT-ORIENTED CODE

SUBMITTED TO THE DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES,

UNIVERSITY OF STRATHCLYDE, GLASGOW

FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY.

By

Alastair Peter Dunsmore

June 2002

Abstract

Since the development of software inspection over twenty-five years ago it has become established as an effective means of detecting defects. Inspections were originally developed at a time when the procedural paradigm was dominant but, with the Object-Oriented (OO) paradigm growing in influence and use, there now exists a lack of guidance on how to apply inspections to OO systems. Object-oriented and procedural languages differ not only in their syntax but also in a number of more profound ways - the encapsulation of data and associated functionality, the common use of inheritance, and the concepts of polymorphism and dynamic binding. These factors influence the way that modules (classes) are created in OO systems, which in turn influences the way that OO systems are structured and execute. Failure to take this into account may hinder the application of inspections to OO code. This thesis shows that the way in which the object-oriented paradigm distributes related functionality can have a serious impact on code inspection and, to address this problem, it develops and empirically evaluates three code reading techniques.

The results from an investigation into the characteristics of “hard to find” defects, in combination with a literature review and an industrial survey, revealed that one of the main difficulties affecting the inspection of OO code was the inherent delocalisation that occurred – OO features distributing closely related information throughout a system. From this, a systematic, abstraction-driven reading technique was developed, focusing on constructing abstract specifications, and evaluated by an empirical study. The results from this led to the development and evaluation of two further reading techniques – one based on a checklist and the other based on a more dynamic approach centered on the route that a use-case takes through a system – along with a refinement of the original systematic technique.

The results indicate that, where practical, object-oriented inspections should be based on teams of inspectors using a combination of at least two techniques. Using a combination of reading techniques, such as those presented in this thesis, seems to offer the potential to deal with recurring defect types, defects that may require deeper insights, and defects that are associated with features of object-orientation that can distribute functionality throughout a software system.

Acknowledgements

I would like to thank Dr. Marc Roper and Dr. Murray Wood for their excellent supervision. Their comments, criticisms, and advice have helped guide and shape the development of this thesis. Without their experience and scathing wit, this thesis would never have reached completion.

Through my time in the department, several individuals have provided encouragement and feedback on the work contained within this thesis. I would like to thank Fraser Macdonald, James Miller, and Douglas Kirk for their comments and encouragement through the course of the thesis. I would also like to thank the support personnel within the department - Ian Gordon, Gerry Haran, and Kenny Forte - for providing technical assistance.

Finally, I would like to acknowledge the encouragement and support given to me over many years by my parents, Helen and Peter, and many friends (Monty, Sam, Gordon, Stuart and Claire). Without it I would have failed long ago. I would also like to say a special thanks to my girlfriend Claire, for her patience and for pushing me over the final finishing line.

The research contained in this thesis was supported by a Graduate Teaching Assistant position with the Department of Computer Science, University of Strathclyde. The support of the Department and the Faculty of Science is gratefully acknowledged concerning travel to conferences. I would also like to add special thanks to Professor Paddy Nixon for help with financial support during the final months of the thesis.

List of Publications

From the work carried out in this thesis there have been a number of publications. These are:

- M. Roper and A. Dunsmore, Problems, Pitfalls and Prospects for OO Code Reviews, *7th European International Conference on Software Testing, Analysis and Review*, EuroSTAR99, 1999.
- A. Dunsmore, M. Roper, and M. Wood, The role of comprehension in software inspection, *Journal of Systems and Software*, 52, pp. 121-129, 2000.
- A. Dunsmore, M. Roper, and M. Wood, Object-Oriented Inspection in the Face of Delocalisation, appeared in *Proceedings of the 22nd International Conference on Software Engineering 2000*, pp. 467-476, June 2000.
- A. Dunsmore, M. Roper, and M. Wood, M., Systematic Object-Oriented Inspection – An Empirical Study, appeared in *Proceedings of the 23rd International Conference on Software Engineering 2001*, pp. 135-144, May 2001.
- A. Dunsmore, M. Roper, and M. Wood, M., Practical Code Inspection for Object-Oriented Systems, in proceedings of the *1st Workshop on Inspection in Software Engineering*, published by Software Quality Research Lab, McMaster University, pp. 49-57, July 2001.
- A. Dunsmore, M. Roper, and M. Wood, Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Inspection, appeared in *Proceedings of the 24th International Conference on Software Engineering 2002*, pp. 47-57, May 2002.

Contents

Chapter 1

Introduction

1.1 Overview

This thesis shows that the way in which the object-oriented paradigm distributes related functionality can have a serious impact on code inspection and, to address this problem, it develops and empirically evaluates three code reading techniques.

Software inspection has, over the last twenty-five years, established itself as an effective and efficient technique for finding defects. Inspections were originally introduced in the late 1970's by Fagan [30] as a *"formal, efficient, and economical method of finding errors in design and code*". The effectiveness of inspections has been established through a large number of controlled experiments and industrial case studies. Fagan [31] reported that it was possible for inspection to find between 60-90 percent of all defects and that the feedback obtained from the inspections was proving useful in helping programmers avoid making the same mistakes. Russell [82] reported savings of nearly 33 hours of maintenance due to every hour spent on inspection.

Inspections, as originally defined by Fagan [30], usually involve four or more people and are made up of several phases: (1) an introduction, where participants are presented with a general overview of the area being addressed; (2) preparation, where individual participants try to understand the artifact under inspection; (3) group inspection, where participants get together as a group and attempt to find as many defects as possible; (4) rework, where defects found are dealt with by the designer or implementor of the artifact; and (5) follow-up, where all issues and concerns are verified as being dealt with.

From their initial use as a code-based technique, inspections are now applied to a wide range of document types including requirements and designs documents [8], [70], [93]. As well as expanding the scope of documentation covered by inspection, the application of the technique and the supporting materials have been refined and honed and there is active interest in continually developing the concept.

In inspection, the focus for detecting defects has moved away from being a group activity to being part of an inspector's individual preparation for the group phase [53], [72], [94]. This refocus has lead to the reading technique (a set of guidelines used by inspector's to acquire a deep understanding of the inspection artifact) becoming a key aspect of the inspection process. Adequate support for inspectors, via the reading techniques, is necessary to help them be efficient and effective in their search for defects.

In spite of their broad application, there is a significant lack of information indicating how inspections should be applied to object-oriented code. Until recently, most of the research carried out in connection with reading techniques, and inspection in general has related to inspections carried out with procedural languages, the predominant paradigm used when inspections were originally proposed. The last ten years have seen the object-oriented paradigm growing in influence and use – particularly since the introduction of C++ and Java. Laitenberger *et al*. [48] commented that "*over the past decade object-oriented development methods have replaced conventional structured methods as the embodiment of software development, and are now the approach of choice in most new software development projects*".

The lack of guidance on how to apply inspections to object-oriented code is disturbing. Object-oriented languages differ from procedural ones in a number of profound ways – the encapsulation of data and associated functionality, the common use of inheritance, and the concepts of polymorphism and dynamic binding – to name but a few. These factors influence the way that modules (classes) are created in object-oriented systems, which in turn influences the way that object-oriented systems are structured and execute. The key features of the object-oriented paradigm may have a significant impact on the ease of understanding of program code and failing to adapt to this paradigm may inhibit the effective application of inspections to object-oriented systems.

This thesis shows that the way the object-oriented paradigm distributes related functionality can have a serious impact on the effectiveness of code inspection and, to address this problem it develops and empirically evaluates three code reading techniques. Each of the three reading techniques address the problem of distributed functionality in different ways, offering the potential to deal with a wide range of defect types.

1.2 Contribution of thesis

The work presented in this thesis makes the following contributions to the area of object-oriented code inspection:

- An investigation of the issues that confound object-oriented code inspection and the identification of three significant issues to be addressed: chunking, reading strategy and dealing with the distribution of functionality (described in Chapter 3 as the problem of ‘delocalisation’).
- The development of three different reading techniques for the inspection of object-oriented code – a systematic, abstraction driven technique, a use-case based approach and a modified checklist – that attempt to address the problems of reading strategy and delocalisation.
- Two controlled empirical experiments to investigate the effectiveness of the three reading techniques developed specifically for object-oriented code.
- A set of lessons, based on the results of the three controlled experiments that can be used to guide current object-oriented code inspection.

1.3 Thesis Outline

The remainder of this thesis is structured in the following way:

Chapter 2: Review of Software Inspection and Object-Oriented Pitfalls

The thesis begins with a review of the relevant literature, discussing the basic principles behind inspection, the different reading techniques that are available to inspectors, the problems caused by object-oriented characteristics, and the current work in the area of object-oriented inspection.

Chapter 3: Investigation of Object-Oriented Code Inspection

An overview of experimentation is presented, highlighting what is considered best practice for preparing and running a software engineering experiment in the context of inspection. This is followed by an experiment investigating the issues surrounding how the object-oriented paradigm impacts on the inspection of object-oriented code. The results from the experiment as well as evidence from a small-scale survey shows that delocalisation is a real problem, and several areas are highlighted that need to be addressed.

Chapter 4: Systematic, Abstraction Based Object-Oriented Code Inspection

A systematic, abstraction based reading technique that attempts to address some of the issues raised from the first experiment is presented and then evaluated by a controlled experiment. The results show no statistical difference in defect detection between systematic and ad-hoc reading techniques, although further analysis of the results show that the systematic technique appears to offer some potential benefits, that with refinement, could help address the problem of delocalisation.

Chapter 5: Further Investigating Reading Techniques for Object-Oriented Code Inspection

Three reading techniques are developed to further investigate the issues concerning object-oriented code inspection - an updated version of the systematic technique, a more traditional checklist technique modified to focus more on object-oriented characteristics, and a use-case driven approach which takes a more dynamic view. This is followed by a controlled experiment that compares the defect detection rates of the three reading techniques. The results suggest that each reading technique has the potential to deal with different defect types.

Chapter 6: Conclusions and Future Work

The final chapter of the thesis contains a summary of the work presented and discusses what lessons can be learned for the practical inspection of object-oriented code. Areas for future work include verification through replication and further refinements to the reading techniques. The conclusion of this thesis is that delocalisation is a significant problem for the effective inspection of object-oriented code, and that where possible, inspections should be based on the use of at least two different reading techniques.

Chapter 2

Software Inspection and Object-Oriented Pitfalls

Much research has been carried out in the area of software inspection since Fagan's original description in 1976. There have been many variations proposed on the traditional inspection process that he first described. Tools have been created to help inspectors find more defects and co-ordinate their efforts in more cost-effective ways. Defect detection aids (e.g. reading techniques) have been defined for different software development artifacts (requirements, code, etc.).

This chapter provides a brief introduction to inspection by describing Fagan's original inspection process. It shows how the focus of detecting defects has moved away from being a group activity to one that is carried out by the individual inspector. This refocus makes the reading technique used by the inspector to help prepare and find defects within an inspection artifact one of the key parts of the inspection process. An overview is presented of the various reading techniques currently available for individual inspectors. This is followed by a review of the literature highlighting the problems that may be caused by object-oriented characteristics, and how these characteristics might have an impact upon code inspection. This chapter concludes with a summary of the current work in the area of object-oriented inspection.

2.1 Inspection

2.1.1 The Inspection Process

Fagan originally defined his inspection process in 1976 [30], later updating it in 1986 [31]. Inspections, as originally discussed by Fagan [30], are *a "formal, efficient, and economical method of finding errors in design and code*". Fagan went on to define an error, or as is now commonly termed, a defect, as "*any condition that causes a malfunction or that precludes the attainment of expected or previously specified results*". As an example, a deviation between a specification and the corresponding code document is a defect.

Inspections can be carried out at many of the stages in the software development process. As well as being used for code documents, inspections are applied to a wide range of artifacts including software requirements, design documents, test plans, and test cases [8], [31], [70], [93].

Code inspections are non-execution based, i.e. the inspector is never allowed to execute or compile the code during the inspection. This allows inspection to be applied to code documents long before tests are designed or even run [36]. It has also been found that if the code is executed and tested before an inspection, the motivation of the inspectors may be reduced and make the inspection process appear redundant [82], [96]. Humphrey [40], in the Personal Software Process (PSP), states that as part of the process to ensure a quality product, inspections should take place before the first compile or test. Taking the opposite view, Gilb and Graham [36] and Strauss and Ebenau [90] consider sending code to a compiler as one of the many different entry criteria that have to be passed before an inspection can begin. The reason for the clean compilation check is that it is cheaper for the compiler (or other automatic tools) to find those kinds of defects, than the more expensive inspector.

In Fagan's original description of inspection [30], there should, under ideal conditions, be four people in an inspection team, each having a specific role. These roles include the Moderator (a competent programmer, sometimes from a different project, to manage the inspection team and offer guidance), Designer (person who produced the program design), Coder / Implementor (person who translated the design into code), and Tester (person responsible for testing the product). In Fagan's original inspection process [30] he defines five main steps (shown in Figure 2.1):

1. Overview – The designer uses this phase to present all the participants involved in the inspection with a general overview of the area being addressed, followed by more specific information on the artifact to be inspected. For code inspections, the overview phase is considered optional.
2. Preparation – This phase is carried out individually. Participants should understand the artifact under inspection using the design documentation. The inspection team are aided in this process by the use of ranked distributions of error types based on recent inspections, as well as checklists containing clues on finding these errors.
3. Inspection – All participants in the inspection group get together. The moderator controls the meeting, making sure that it stays focussed, so that it does not get out of hand or stray off course. All related documentation should be available during the

inspection. With the design of the artifact under inspection understood (in the previous preparation phase), the main objective in this phase is to find defects. This occurs as the "reader", chosen by the moderator (usually the coder) takes the team through the inspection artifact. Once a defect is found, no attempt should be made by the inspectors to find a solution. Defects are noted by one of the group members given the task of being meeting scribe (either the tester or someone with no other task).

4. Rework – All the defects noted in the inspection report from the previous phase are resolved by the designer or implementor.
5. Follow-up – All issues and concerns are verified as being followed-up. If more than 5% of the material inspected has in some form had to be reworked, the inspection team should regroup and carry out a full re-inspection of the material.

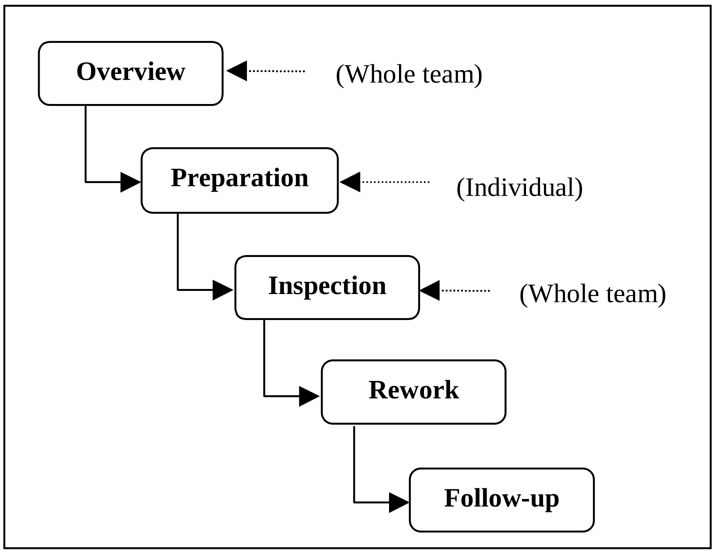


Figure 2.1 – The five steps in Fagan's original inspection process

Since Fagan's original inspection process, there have been many variations attempting to improve the performance of inspections. Active Design Reviews [69] were originally created to ensure complete coverage of design documents and advocate several small, focused inspection meetings rather than one large meeting involving a lot of people. In each of these smaller meetings, inspectors are assigned a specific role to look for different types of defect. In N-Fold Inspections [83] not one, but many parallel inspections are performed by different teams on the same artifact. The assumption is that a single inspection team will only find a fraction of the defects, and that multiple teams will not significantly duplicate each other's efforts. Phased Inspections [45] divide the normal inspection into several smaller phases. These phases can be carried out by one or more inspectors. Each phase focuses on one specific type of defect (compared to more traditional inspections, which look for all types of defect in one big inspection). If more than one inspector is involved, they meet to create one definitive defect list. Phases are carried out in sequence, meaning that the next phase is not reached until the previous one has been completed. Sample-Driven Inspections [92] is a method designed to reduce the effort during an inspection session by concentrating the inspection effort on the software artifacts that contain the most defects. The defect searching is divided into two parts. A pre-inspection occurs where samples of the artifacts are inspected to estimate which artifacts contain the most faults. Secondly, the main inspection is carried out on the selected artifacts. These alternative processes have varied such elements as the number of steps in the inspection process, the number of inspectors, and the roles of inspectors. Although each variation has made alterations to the inspection process or altered characteristics of the phases, the inspection phases of preparation, inspection, and rework/follow-up from Fagan's original description have remained [50].

There have been many reports on the successes achieved through the use of inspections. Fagan [31] commented that inspection was detecting between 60 to 90 percent of defects. Ackerman *et al*. [1] reported that inspections were two to ten times more efficient at defect removal than testing. Russell [82], based on 2.5 million lines of high-level code, found that if inspection was correctly implemented, then approximately one defect was found for every man-hour invested. Russell claims this was two to four times faster than detecting defects by testing. Reports by Weller [96], Grady and Slack [37], have also supported the use of inspection, detailing improvements to the process and suggestions for achieving widespread use.

In Fagan's original inspection process [30], the preparation phase was used by inspectors to obtain an understanding of the inspection artifact and the inspection phase was used by the inspectors as a group to carry out defect detection. A series of recent empirical studies investigating the group aspect of the inspection process have cast doubt on its relevance as a focus for defect detection. Votta [94] suggests that inspection meetings are no longer required since the number of extra defects discovered in the meeting over those found in the individual phase is relatively small (average 4%), and they are not cost effective due to the time delay in preparing, organising, and holding the inspection meetings. Meetings should be replaced by either small deposition meetings (used to collect reviewers’ findings and comments), or defect lists should be collected by other verbal or written media (e.g. electronic mail, telephone). It was found that meetings help reduce the number of false positives (potential defects which turn out not to be actual defects). Land *et al.* [53] found that the strength of inspection meetings is not in finding defects, but discriminating between true defects and false positives. They found that only a small number of extra defects were found by inspectors when working in a group. Porter and Johnson [72] found that far more issues are generated by individual defect detection compared to group-based defect detection, but this comes at the cost of higher rates of false positives and defect duplication. The current goals of the group aspect of inspection are now for the inspectors to agree upon a final list of defects based upon those found individually, and to reduce the number of false positives in the final report [51]. The main focus for the preparation phase of inspection is now the detection of defects [51], [73].

Porter and Votta [74] found that defect detection results have less to do with the particular inspection process used, and have more to do with the techniques and technology supporting individual inspectors. Giving support to individual inspectors to find defects may increase their effectiveness.

With the re-emphasis of the defect detection part of the inspection process on the individual preparation phase, there has been a shift in inspection research. Basili [9] pointed out that reading was by far the most important activity for successful individual defect detection. Basili also highlighted the lack of research examining the technologies that underlie the reading process. One reason for this lack of research was that until recently, much of the research into inspection has been focused on the inspection process [73], [79]. Adequate support for the defect detection activity of inspectors (i.e. reading strategies) has the potential to dramatically improve the effectiveness and efficiency of inspection [51]. The more the inspector can understand the material to be inspected, the greater the chance of finding defects [79].

Although the most recent work on inspection reading techniques has focused on design and requirements documents, in industry the inspection of code documents is still predominant [50]. Laitenberger *et al.* [51] concludes that this makes the improvement of reading techniques for code documents a high priority. The next section presents a summary of the reading techniques and looks at how they attempt to help the inspector find defects.

2.1.2 Reading Techniques

Laitenberger and DeBaud [50] described a reading technique as *a "series of steps or procedures whose purpose is for an inspector to acquire a deep understanding of the inspected software product*". In Fagan's original inspection process he suggested the use of checklists [30]. As well as checklists, another popular technique in industry has been ad-hoc inspection [27], [36]. With the emphasis of defect detection being placed on the preparation phase of inspection [51], [73] and a realisation that reading is important for defect detection, there has been a renaissance in the development of reading techniques. The following describes some of the more prominent reading techniques currently available.

2.1.2.1 Ad-hoc

One of the simplest reading techniques, ad-hoc, provides no support for inspectors, i.e. no guidelines or direction. Inspectors have to rely on their own knowledge and experience, reading the inspection artifact, whether they are specifications or code, in their own preferred way. Although the ad-hoc approach offers no guidance to inspectors, it is considered to be a reading technique [50], [71].

A strength of the ad-hoc technique is that more experienced inspectors have the freedom to use their knowledge and abilities to find defects, free from any technique overhead that may intrude upon their thinking. The main weakness of the ad-hoc technique is that with no support, the performance of the less experienced inspectors may suffer, since they do not have the experience to guide them.

2.1.2.2 Checklist

Checklists, which have been around since the early use of inspections in the late 70's, are straightforward to use and offer stronger guidance to inspectors than ad-hoc reading. They are based upon a series of specific questions that are intended to focus the inspector's attention towards common sources of defects. The questions in a checklist are there to guide the inspector through the document under inspection. To make it clear that a potential defect has been found, the questions are phrased in such a way that if the answer is 'No', then a potential defect has been discovered. According to Gilb and Graham [36] and Humphrey [40], checklists should be based on localised historical information and should not be general checklists obtained from elsewhere as they can lose their relevance. An excerpt from an example C++ code review guideline and checklist by Humphrey [40] can be seen in Figure 2.2. Checklists, along with ad-hoc reading are still thought of as the most frequently used defect detection methods [36], [73]. Checklists have been used to inspect many different documents, including design, specification, and code.

Although checklists have been well promoted [31], [40], there are several weakness which have been identified. Laitenberger *et al*. [50] summarised a list of the weaknesses of the checklist technique from the literature. Firstly, that the questions are often too general or based upon checklists created from the defect experience of others. Similarly, Tervonen [91] commented that one of major problems facing checklists is their generality, that they are not sufficiently tailored to a particular development method or phase in a specific project. Second, instructions guiding inspectors on how to use a checklist are rarely available, i.e. it is often unclear when and based on what information an inspector is to answer a particular checklist question. Finally, the questions of a checklist are often limited to the detection of defects which belong to particular defect types. Since the defect types are based on past information [19], inspectors may not focus on defect types not previously detected and, therefore may miss whole classes of defects (a problem only slightly reduced by the constant revision that should occur with checklists).

Initialisation	Check variable and parameter initialisation: • At program initiation • At start of every loop • At function/procedure entry
Calls	Check function call formats: • Pointers • Parameters • Use of '&'
Strings	Check that all strings are • identified by pointers and • terminated in NULL.
Pointers	Check that: • pointers are initialised NULL, • pointers are deleted only after new, and • new pointers are always deleted after use.
Output Format	Check the output format: • Line stepping is proper. • Spacing is proper.
Logic Operators	Verify the proper use of ==, =, ||, and so on. Check every logic function for proper ().

Figure 2.2 - C++ Checklist, from Humphrey [40]

2.1.2.3 Step-wise Abstraction

The step-wise abstraction reading strategy offers more structured and focused instructions on how to read code. The technique was based on the step-wise abstraction technique of reading developed in the late 70's by Linger, Mills and Witt [59]. In step-wise abstraction, the aim is to start with the simplest components in the code, understand them, and abstract out a higher level description of their functionality [3]. This process is repeated, combining higher and higher levels of functionality, until a final description of the code is obtained. This final description is then compared with the original specification. This way any differences between the original specification and the derived specification highlight potential defects. Stepwise abstraction has been most commonly used as a code reading technique by the Cleanroom community [84] (the Cleanroom development method is a technical and organisational approach to developing software with certifiable reliability).

Based upon evidence from the literature, Laitenberger *et al*. [51] believed that inspectors utilising the step-wise abstraction technique were forced into a more rigorous examination of the code than using either the ad-hoc or checklist reading techniques.

2.1.2.4 Scenario-Based Reading

The scenario reading strategy was created by Porter *et al*. [70] to address a perceived lack of effectiveness in the use of ad-hoc and checklist methods for Software Requirements Specifications (SRS). The work builds on the inspection process Active Design Reviews by Parnas and Weiss [69], who argued for the need for different and specific roles for inspectors to systematically inspect a document. Porter *et al*. described a scenario as a "*collection of procedures that operationalise strategies for detecting particular classes of defects*". Each inspector is given one scenario, which differs from the scenarios given to the other inspectors in the inspection team. Each scenario contains a set of questions and instructions informing the inspector how to perform the inspection of the SRS. Multiple inspectors are required to obtain a reasonable level of coverage from the document. The scenarios generated by Porter *et al*. [70] were derived from available defect classes.

The success of this technique relies heavily on the effectiveness of the designed scenarios. Several variations on the scenario approach have been developed, each varying the way the scenarios are created. In defect-based reading by Porter *et al*. [70], the scenarios are derived from defect classes with a set of questions the inspector has to answer. For scenario-based reading by Cheng and Jeffrey [18], the scenarios are based on Function Point Analysis (scenarios are developed around a software system defined in terms of its inputs, files, enquiries, and outputs). In perspective-based reading by Basili *et al*. [8], the inspection artifact is inspected from the perspective of different stakeholders. Each of these reading techniques provide a generic process for inspecting requirements documents, although the material generated by the processes for use in inspections are target specific (to a particular development environment).

The last of these techniques, Perspective-based reading, has continued to be refined and has been implemented not just for requirements documents but for code documents as well.

2.1.2.5 Perspective-Based Reading

Perspective-based reading (PBR), first presented by Basili *et al*. [8], evolved from the work carried out on scenarios. PBR, compared to the Scenario technique, offers a more detailed set of instructions (scenarios) for inspectors. The perspective-based scenarios are an algorithmic set of instructions informing inspectors how to read an artifact under inspection. Inspectors understand the artifact by constructing an appropriate abstraction defined by the scenario. Laitenberger and DeBaud [47] claim that a focused understanding of the document obtained through the use of PBR should be more effective than either an ad-hoc or a checklist based reading technique. Ad-hoc and checklist based reading techniques are thought of as non-systematic in nature [73]. They do not offer a set of concrete reading instructions, meaning that inspectors' experience has a significant impact on the number of defects found [47].

The PBR technique continues to be refined, giving better instructions on the creation and content of scenarios [48]. A PBR scenario contains three parts. The first explains to inspectors their interest/perspective on the inspection artifact. The second part consists of a set of activities that inspectors have to perform. This allows them to extract the required information out of the inspection artifact. In the final part, inspectors then apply a series of questions to this information to verify its correctness. An example of a code scenario for the C programming language is shown in Figure 2.3, created by Laitenberger *et al*. [51]. In an inspection, each inspector has a different scenario to allow the artifact to be looked at from different views, e.g. analyst, maintainer, tester, etc. By following the scenario the inspectors should build up an understanding of the artifact. Although the early work on PBR was carried out on requirements documents [8], some of the more recent work has focused on C code documents [47], [51].

Basili *et al*. [8] found through experimentation that less experienced inspectors learned to apply PBR better, and that the perspectives helped them focus more whereas more experienced inspectors were more likely to revert to their more traditional or previously learned techniques.

Tester Scenario

Assume you have the role of a tester. As a tester you have to ensure that the functionality implemented in the code is correct.

In doing so, take the code document and determine the functions that are implemented in this code module. Determine the dependencies among these functions and document them in the form of a call graph.

Starting with the functions at the leaves of the call graph, determine for each function, a set of test cases that allow you to stimulate the operation of the function. The set of test cases should allow you to check each branch of the function as well as the loops. Document some of the test cases.

Assume you are executing the function with your test cases as input values (mental simulation). Verify whether each function behaves according to its specification and the comments given in the code. If differences occur, check whether there is a defect or not. Document each defect you detect on the defect report form.

While following the instructions, ask yourself the following questions:

1. Do you have the necessary information to identify a test case (e.g., are all constant values and interfaces defined)?
2. Are branch conditions used in a correct manner?
3. Can you generate test cases for each branch and each loop? Can you traverse all branches by using specific test cases?
4. Is allocation and de-allocation of memory used correctly?

Figure 2.3 – The tester scenario for C code documents, from Laitenberger *et al*. [51]

An experiment by Laitenberger *et al*. [51], investigated the effectiveness and cost per defect ratio of PBR compared to checklists for C code documents. The results showed that two-person inspection teams were more effective using PBR than checklists. Applying PBR was found to increase subjects understanding of the code, but was found to require greater effort from inspectors. This improved understanding was also found to have helped to reduce the cost of defects for PBR compared to checklists during the meeting phase. With a greater understanding in the meeting, it took less effort on the inspectors' behalf to explain the defect they had found to the other inspectors, as well as taking less effort to resolve false positives. It should be noted however, that the checklist used during the experiment was a general one, based upon an existing checklist [66] and books on C programming [23], [43]. This goes against the currently available advice [36], [40], which states that checklists are most effective when based upon historical data.

Although most of the experiments investigating the effectiveness of using PBR have been positive, there has recently been one experiment (based upon a lab package by Basili *et al*. [10]) investigating its effectiveness and efficiency with relation to requirements documents [78]. The results showed that there was no significant difference in the defect coverage of the three perspectives, suggesting that a combination of multiple perspectives may not result in a higher defect coverage compared to reading with only one perspective. This contradicts the earlier work on PBR. Regnell *et al.* [78] provide no other reasons for the results, other than to highlight certain threats to the validity of the experiment. The threats included the setting may not be realistic, the perspectives may not be optimal, subjects may not be motivated or trained enough, and the number of subjects may be too small.

2.1.2.6 Summary

Reading techniques have evolved from offering no support and minimal guidance to inspectors into detailed task driven processes that encourage inspectors to attain a good understanding of the artifact under inspection. More recent reading techniques have also introduced the notion of inspecting artifacts from different views (perspectives). This allows inspectors to focus on different aspects and different defect types in greater detail.

The increased understanding promoted by recent reading techniques is achieved through clear, unambiguous instructions that guide the inspector in extracting and querying the required information from the inspected artifact. It is the development of this good understanding of the code that is key to a successful inspection. The main drawback to these more process driven techniques is the extra work required to be done by the inspector.

2.2 Object-Oriented Problems and Pitfalls for Inspection

The object-oriented paradigm has gained widespread acceptance [17] and, it has been argued, has delivered many benefits to the programmer such as better structured and more reliable software for complex systems, greater reusability, more extensibility, and easier maintainability [44]. With these claimed successes, there have also arisen new problems to be tackled. In 1994, Jones [41] listed some of the gaps in information about the object-oriented paradigm. One of those gaps was in the area of inspection. Jones noted that "*Since formal inspections are the most effective known way of eliminating software defects,* *software quality assurance personnel are anxiously awaiting some kind of guidance and quantitative data on the use of inspections with object-oriented projects*".

There is a significant body of literature developing that suggests that the characteristic features of the paradigm can make object-oriented code more difficult to understand compared to the procedural equivalent – an issue that has direct impact on code inspection. Much of this literature centres on experience gathered from the software maintenance domain. The problems encountered in maintenance can apply equally to the task of inspection - both require sections of code to be read and understood (it is assumed that inspection performance is closely related to comprehension – see [26], [51], [79]).

According to Gamma *et al.* [35], the structure of an object-oriented program at run-time is vastly different to that of its code structure, *"In fact, the two structures [run-time and compile-time] are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice-versa*." Where the code structure is frozen at compile-time, the run-time structure consists of rapidly changing networks of communicating objects. This makes it very difficult to understand one from the other.

Dependencies exist in all code, but their number are increased by object-oriented languages [17], [97]. Wilde and Huitt [97] described a dependency in a software system as "*A direct relationship between two entities in the system X → Y such that a programmer modifying X must be concerned about possible side effects in Y*". Wilde and Huitt suggested that using polymorphism and inheritance hierarchies dramatically increases the kinds of dependencies that need to be considered. Some of the dependencies they highlighted include Class-to-Class, Class-to-Methods, Class-to-Message, Class-to-Variable, Method-to-Variable, Method-to-Message, and Method-to-Method. Chen *et al*. [17] described three kinds of dependencies found in object-oriented languages, message dependence (relationship between a method and its callers), class dependence (inheritance, aggregation and association relationships) and declaration dependence (relationship between classes (types) and objects (variables)).

Dynamic binding is a specific example of a characteristic of object-oriented programs that increases the complexities and dependencies in a program. This concept, closely associated with polymorphism, involves not knowing the type of a particular object referenced by a variable, as this is only determined at run time [14], [61]. When a method invocation occurs, only at run time can the type of an object be correctly identified. All the associations created through the use of polymorphism and dynamic binding usually mean that more than one class needs to be looked at (especially in the case of a class which is part of a deep inheritance hierarchy) in order to fully understand how one small fragment of code works. Wilde and Huitt suggested that tracing these dependencies is vital for effective software maintenance [97]. Lejter *et al*. [55] claimed that dynamic binding (along with inheritance) made object-oriented programs much more difficult to maintain and understand. This view is also supported by Crocker and von Mayrhauser [20].

The structure of object-oriented programs differs from that of conventional programs [97]. Method sizes may be very small as a natural consequence of good object-oriented design [57], [98]. Daly *et al*. [22] found that unconstrained use of inheritance may result in understanding difficulties. When investigating the difficulties experienced programmers encountered learning and using the Smalltalk programming language, Nielsen and Richards [67] found that the distributed nature of the code caused problems when attempting to understand a system. Together with inheritance, this distribution may result in traversing up and down inheritance hierarchies and across class boundaries in an attempt to locate where the work is carried out and build up a complete understanding of the task. This problem is illustrated in Figure 2.4.

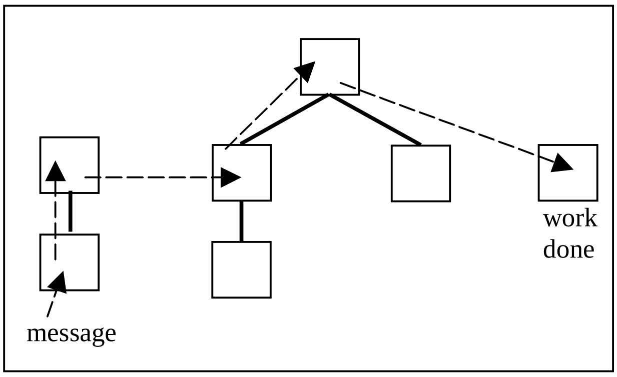


Figure 2.4 - Chain of message invocations

Many of the problems that have been mentioned have also created difficulty for other areas of software engineering outside of software maintenance, such as comprehension [54], component reuse [32], testing [65], [42], and visualisation [54]. Each area has had to re-evaluate how it achieves its objectives, and in many cases redesign its processes. Binder [13] in his review of testing for object-oriented software highlighted that most believe the features of object-oriented systems (inheritance, polymorphism, abstract super classes, encapsulation) will require the development of new approaches to be able to achieve adequate levels of testing.

2.3 Current state of Object-Oriented inspection

With the rise in popularity of object-orientation, the research community has turned to adapting inspections to this new paradigm and its particular artifacts. So far, the work carried out has focused on the inspection of object-oriented requirements and design artifacts. Although some initial work has been positive, there has been a lack of research regarding how the key features of the object-oriented paradigm may impact on the inspection of object-oriented code.

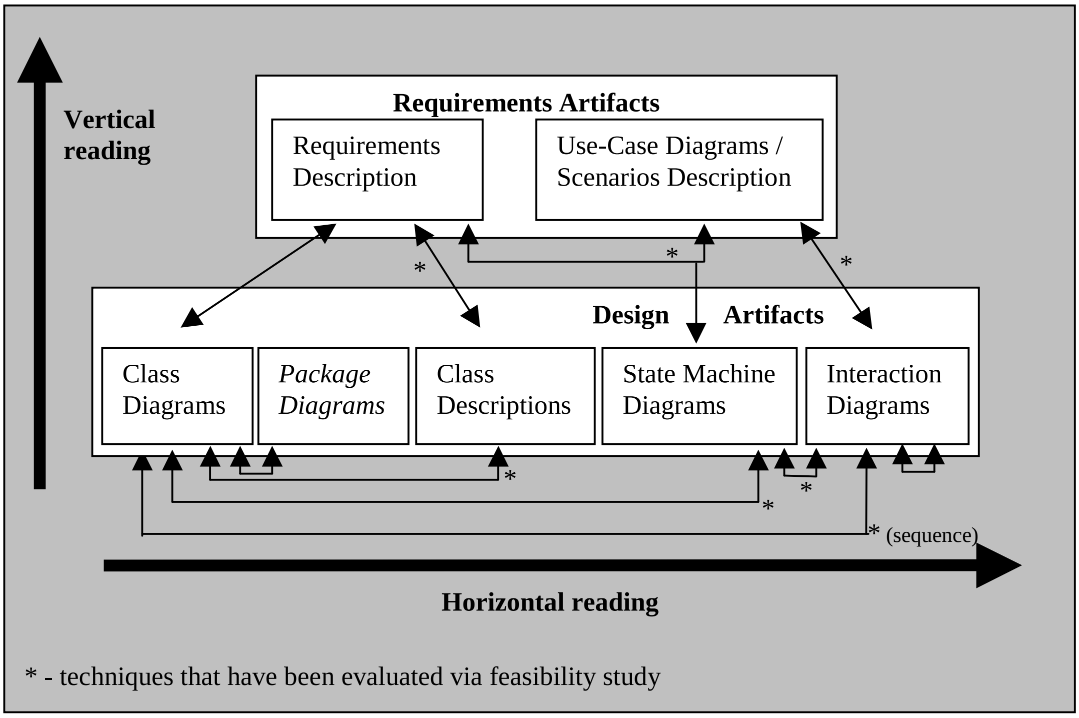


Figure 2.5 – Reading techniques in Traceability-Based Reading (TBR), from Travassos *et al*. [93]

Travassos *et al*. [93] found that there was a lack of guidance on how to inspect object-oriented design documents and carried out a preliminary investigation. The main focus was on designs described by UML diagrams. They developed a technique called Traceability-Based Reading (TBR) that evolved from the experience gathered from the development of reading techniques for requirements documents [70]. TBR is a two step process. The first step involves the correctness and consistency checks on requirements documents that have traditionally occurred. This is described as horizontal reading. The second step is described as vertical reading, and differs from the traditional process, in that requirements documents are compared with design documents to ensure consistency. This is illustrated in Figure 2.5, based on a diagram by Travassos *et al*. [93].

An experiment carried out using TBR found encouraging, but not conclusive results. Horizontal and vertical reading were found on average to highlight different types of defect. Vertical reading found slightly more defects concerning omission and inconsistency (between diagrams and requirements), where horizontal reading found more defects concerning ambiguity and inconsistency (between diagrams). An important finding was that the technique forced more focus on semantic information (an understanding of the meaning of the document), similar to the focus encouraged by the scenarios of PBR. In its current state, the technique relies too much on syntactic information, making sure that certain words and attributes in one diagram appear in their correct location elsewhere. Another drawback is that the technique was found to be time consuming.

Laitenberger and Atkinson [48] presented an adaptation of Perspective-Based Reading (PBR) for any object-oriented development artifact. They provided a generally applicable definition of the technique, describing instructions on how to generate PBR scenarios. An experiment [49] was carried out to investigate the effectiveness of PBR for UML design documents in comparison to checklists. The results of the experiment showed that PBR scenarios help improve inspectors understanding of the inspection artifacts. This was found to reduce the cost of defects in the group phase (as a collation exercise) for PBR in comparison to checklists. The checklists used were designed along the lines discussed by Chernak [19], but the questions, due to the lack of other such checklists for object-oriented design documents that could be reused, were developed from scratch.

The majority of research carried out in the area of object-oriented inspection has so far been aimed at the development of reading techniques to help inspectors find defects in requirements and design documents. These techniques have tried to address a perceived lack of reading guidance, but have not fully investigated how the key features of the object-oriented paradigm impact upon code inspections. Tervonen [91] has found that existing object-oriented checklists are focused primarily on design issues and are therefore not suitable for code inspection.

It is generally understood that the earlier an inspection occurs, the cheaper the cost of repairing the defect [37], [85]. This is what has lead to the greater emphasis in developing reading strategies for early development artifacts. However, as highlighted by

Laitenberger *et al*. [50], code inspections are still the most commonly occurring in industry and, as such, this makes the improvement of reading techniques for code documents a high priority [51]. This takes on even greater importance when taking into consideration the lack of research regarding how the key features of the object-oriented paradigm may impact on the inspection of code.

2.4 Summary

Inspections are an effective method used to find defects in many different documents generated throughout the lifetime of a software project. Recently, the focus for detecting defects has moved away from the group inspection activity. Instead, the focus for detecting defects is the preparation phase, where the individual inspector reads the artifact in preparation for the group phase (which is now used for defect collation).

With the focus for detecting defects in inspection moved to the preparation phase, the reading technique used by the inspector to help prepare and find defects within an inspection artifact has become one of the key aspects of the inspection process. Adequate support for inspectors is necessary to help them be as efficient and as effective as possible.

Reading techniques have evolved from offering no support and minimal guidance to inspectors (e.g. ad-hoc and checklist) into detailed task driven processes that encourage inspectors to attain a good understanding of the artifact under inspection (e.g. scenarios and perspective-based reading). It is the development of this good understanding of the code that is key to helping inspectors increase their effectiveness.

Within the last decade, the object-oriented programming paradigm has grown both in influence and use. Many of the key characteristics of object-oriented languages - inheritance, dynamic binding, polymorphism, and small methods complicate matters. Many of these characteristics lead to closely related information being distributed throughout the code, significantly impacting upon the ease of understanding.

To date, much of the work carried out investigating the inspection of the object-oriented paradigm has concentrated on requirements and design documents. None of this work has addressed the issues regarding how the key features of the object-oriented paradigm may impact on the inspection of code. Currently available reading techniques were developed at a time when the procedural paradigm was dominant, meaning they may not address effectively the features of the object-oriented paradigm.

With code inspections dominant in the software industry, there is a clear need to investigate the effect of the object-oriented paradigm. This is likely to have an important impact on the development of future code reading techniques.

Chapter 3

Investigation of Object-Oriented Code Inspection

This chapter presents an investigation of how the object-oriented paradigm impacts on the inspection of object-oriented code. It begins with an overview of experimentation and highlights what is considered best practice for preparing and running a software engineering experiment in the context of inspection. A controlled experiment is then presented that investigates how the object-oriented paradigm impacts on the inspection of object-oriented code. A detailed analysis of the characteristics of the 'hard to find' defects, together with the results of a small-scale survey of software engineering professionals, suggests that 'delocalisation' - the distribution of closely related information throughout the code - is a major problem. The chapter concludes by looking at how current reading techniques for code inspection deal with the problem of delocalisation, and highlights chunking, reading strategy, and 'localising the delocalisation' as areas that need to be addressed.

3.1 Experimental Software Methodology

Empirical research in the context of software engineering is conducted to help evaluate, predict, understand, control and improve the software development process or product [5].

In software engineering experimentation there are a variety of methods a researcher can utilise in order to gather information and evaluate their notions and hypotheses. Methods available include interviews, questionnaires, observation, case studies, and controlled experiments [24].

A significant amount of research has been carried out investigating various aspects of inspection, e.g. process [45], [69], reading strategy [51], [70], tool development [62], [77], numbers of inspectors [73], [96], the need for group meetings [72], [94], etc. Much of this research has been achieved by designing and running empirical studies.

Lott and Rombach [60] detailed an experiment characterisation scheme that can be used as a basis for empirical software engineering research. The scheme permits the comparison of results from similar experiments and establishes a context for cross-experiment analysis. This issue has become more important with the ever-growing amounts of experimental data and the desire to compare experimental results [63]. The scheme is also a good guide to designing experiments. There are four parts to the experimental characterisation scheme. They are, (1) the goals and hypotheses that motivate an experiment, (2) the plan to conduct the experiment, (3) the procedures to be used during the experiment, and lastly, (4) the results which detail the raw data collected during the experiment and any analysis carried out. The following briefly summarises each:

1. Goals and Hypotheses

The goals and hypotheses should be used to quantify the expected outcomes of an experiment, and be used to aid in the design and running of the experiment [5]. To help focus the development of the goals and hypotheses the Goal Question Metric (GQM) paradigm can be used as described by Solingen and Berghout [87] (originally developed by Basili and Weiss [4] and augmented by Basili and Rombach [6]). The GQM shifts the emphasis away from metrics to goals. The goals create a focus for the important issues of an experiment. These goals are then specified in more detail by defining questions, which in turn suggest the appropriate metrics to be measured. With the goals for an experiment stated explicitly, the data collected and the evaluation of that data are based on well-specified rationale.

2. Plan

The plan for the experiment details all the design decisions made. The plan includes the goals and hypotheses already generated, along with such elements as subjects used, material used, e.g. code, diagrams, and defects inserted in code. All of these have to be justified within the frame of the goals of the experiment. The variables being investigated by the experiment are usually detailed in the plan. There are two kinds of variables, independent and dependent. Independent variables are those that are believed to have an influence on the result of the experiment, e.g. reading strategy or code to which reading strategies are applied. Dependent variables measure the effects of the manipulation of the independent variables, e.g. number of defects found by subjects or time taken.

A very important part of the experimental plan is the validity section. This details two kinds of validity, internal and external. An empirical study can suffer from influences which may affect the experimental variables without the knowledge of the researcher (internal validity), e.g. selection effects, plagiarism, subjects' enthusiasm, or learning effect. Threats to external validity limit the ability to generalise any results from an experiment to a wider population, e.g. representative subjects, code and defects used, or process used.

3. Experimental Procedures

The procedures to be used during an experiment include details of how the experiment will proceed. A timetable is given describing what events will occur, e.g. lectures, training, assessments, and when they will occur. Also detailed here is the material that will be available for each part of the experiment, e.g. code documents, forms, and questionnaires.

4. Results

The results section contains a detailed description of the raw data collected during the experiment, the results of any statistical analysis based on the raw data, and an interpretation of these results.

Having a detailed collection of resources allows for repeated experiments (replication). Replication allows for the verification and validation of previous results, building up a supportive body of knowledge and understanding, which can be used to justify the usefulness of techniques and new methodology. As discussed by Basili *et al.* [7], "*In examining and adapting reading techniques, we go through a systematic process of evaluating the candidate process and refining its implementation through lessons learned from previous experiments and studies*".

In many cases students in the university environment are used to evaluate initial theories and techniques [39]. There are several reasons for this, (1) they are a relatively cheap resource, (2) there are usually a sufficiently large number of students, and (3) it is cheaper for a technique to fail in the lab using students than out in industry with industrialists. This style of development follows an iterative approach. A technology can be tested with students to explore initial ideas and theories. The technology can go through several revisions, each time being refined by the results of the previous experiment before being evaluated in industry.

Daly [21] proposed that a multi-method approach should be taken towards empirical software engineering to address the challenges created by the human element and problems of experimental validity. An example would be to use different techniques, e.g. questionnaires and interviews, to identify hypotheses or validate results with different subject groups, e.g. industry.

3.2 An Experiment investigating Object-Oriented Code Inspection

3.2.1 Introduction

The object-oriented programming paradigm has grown both in influence and use. Many of the key characteristics of object-oriented languages - inheritance, dynamic binding, polymorphism, and small methods complicate matters. With a lack of research regarding how the object-oriented paradigm may impact on the software inspection process, a controlled experiment was designed to investigate how the characteristics of object-oriented code effect code inspection.

The experiment was motivated by a question posed by Laitenberger and De Baud [50]: "*How can inspection, a static analysis process, ensure the quality of artifacts involving the use of such non-static features as dynamic binding?*". What are the issues that arise when reading and understanding object-oriented program code with the aim of detecting defects? What are the 'hard to find' defects, and why are they so hard to find?

3.2.2 Experimental Goals and Hypotheses

The goal of the experiment was to investigate the possible link between defect characteristics and ease of detection. To do this, the experiment was designed to be a qualitative investigation. This did not require the generation of specific hypotheses, since no statistical analysis was required.

3.2.3 Experimental Plan

The experiment was based on a code inspection exercise that was solely concerned with the effort of the individual - no group component was carried out. As the experiment focused on the number of actual defects found by subjects, the number of false positives generated by the inspectors was not investigated (defects listed by subjects during the inspection that were not actual defects).

To inspect the code, subjects used the ad-hoc reading strategy. "*Ad-hoc does not mean that inspection participants do not scrutinise the inspected product systematically. The word 'Ad-hoc' only refers to the fact that no support is given to them. In this case defect detection fully depends on the skill, the knowledge, and the experience of the inspector which may compensate for the lack of reading support*" [50].

The code inspection was paper based, no tool support was provided. Aids such as checklists, other reading strategies or inspection tools were not used because they may have introduced confounding factors into the experiment, making any analysis more difficult.

A copy of all the material used in the final experiment can be found in Appendix A.

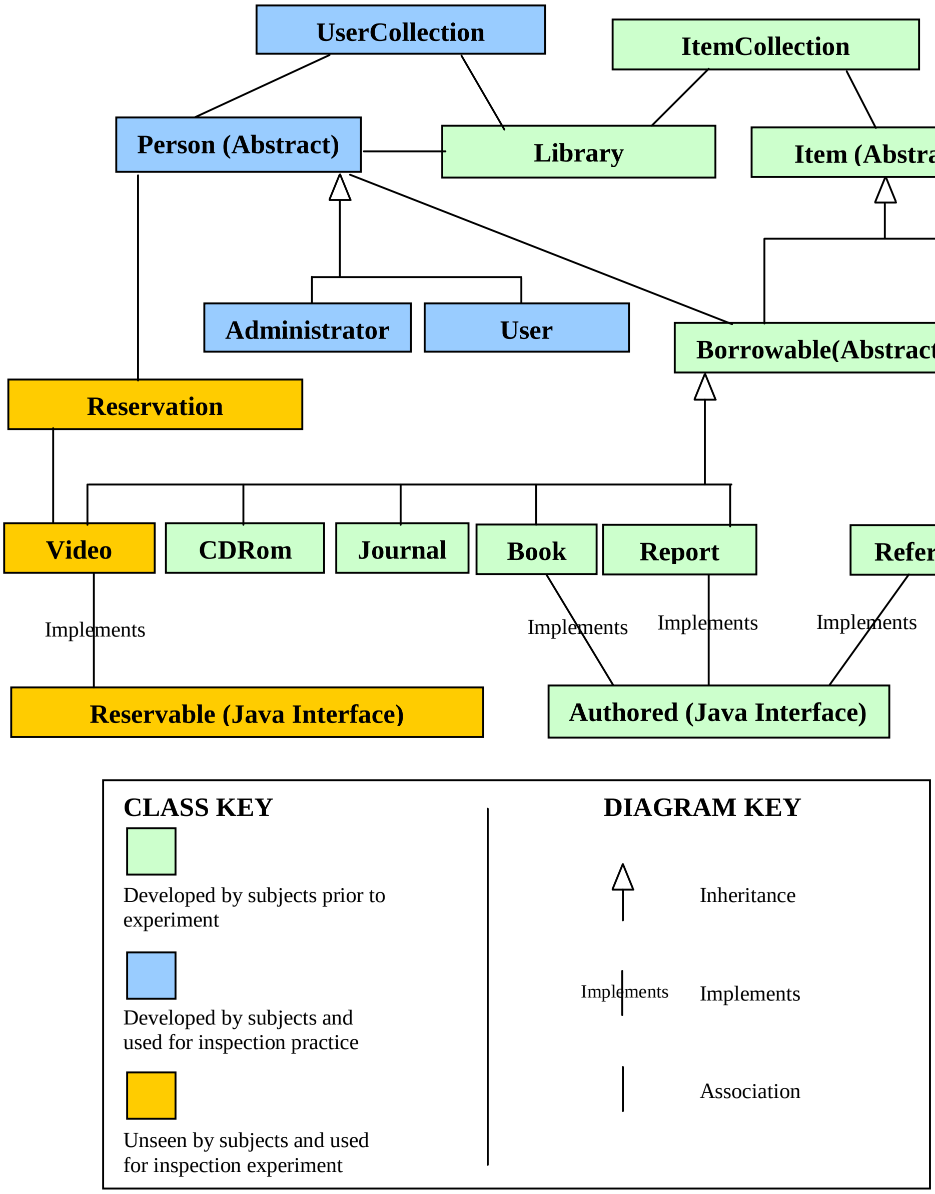


Figure 3.1 – Class hierarchy diagram for library system

Subjects

Subjects were participants in a 3rd year Honours Computer Science Software Engineering course run at the University of Strathclyde. 47 subjects were participating in the class. Subjects had previous experience with the programming languages of Scheme, C, C++, Eiffel, and Java (the three months preceding the experiment). The subjects had limited knowledge of Software Requirement Specification (SRS) document inspection, and no experience with code inspections.

Prior to the experiment, subjects were given a problem statement describing a simple library system (the original problem statement can be found in Appendix A.1). The library system has a number of different items that can be borrowed, e.g. books, reports, CDRoms and also contains reference material that cannot be borrowed. Subjects were given six weeks to derive a semi-formal specification and design for the system. Once this was completed, subjects were provided with a design prepared by the course lecturer (the class hierarchy for the system is shown in Figure 3.1). From this, subjects were given a further 6 weeks to code the library system using Java. The experiment took place after the coding of the system was complete.

For the inspection exercise, two groups of subjects were created, group A and group B, where subjects in each group were of approximately equal ability (based on previous programming courses). This was done to allow two groups of defects to be seeded, increasing the number of defects investigated.

Code

The language chosen for the experiment was Java. This was for two reasons, (1) the language had to be object-oriented, (2) the subjects had been using Java in the months preceding the experiment in the software engineering course.

The code presented to the subjects for inspection was approximately 200 lines in length. This was chosen as a maximum limit. Fagan [31] suggested that a maximum of 125 non-commentary source statements per hour are read. Weller [96], from information gathered from over 400 inspections, suggested no more than 200 lines of code per hour. Gilb and Graham [36] suggested at most one and a half pages (approximately 90 lines of code) per hour. Although there is no clear consensus on inspection rates, the literature generally agrees that inspecting too much code reduces the effectiveness of code inspections. A maximum of 100 lines of code per hour was chosen for the inspection, bearing in mind the subjects were 3rd year Computer Science students, not professional inspectors.

The experiment was split into two phases, practice and experiment proper (these are described in Chapter 3.2.4, Experimental Procedures). For the practice session of the experiment subjects were presented with part of the system that they had previously coded but that was now written by the class lecturer. For the actual experiment the library system was extended and extra functionality, written by the class lecturer, added. This extra functionality entailed the implementation of a video class and the need for a reservation class (in order to be able to reserve a video for a specific date). The subjects had not previously seen any code or design for the extension.

Defects

The experiment required a selection of defects to be seeded in the code. The review of the literature presented in Chapter 2.2 highlighted potential problem areas that may be used as the basis for some seeded defects in object-oriented code: dynamic binding, polymorphism, small methods, and inheritance. There is, however, currently very little material in the literature discussing object-oriented code inspections or typical defect categories for object-oriented code.

Duncan *et al.* [25] carried out a review of testing techniques and taxonomies which highlighted that only a small amount of work had been carried out in the area of fault classification for the object-oriented paradigm, and that there was a lack of experiments to show what faults were commonly occurring. Extrapolating from a category of classification for non-object-oriented code, Duncan *et al.* suggested that potential sources of object-oriented faults might be in instance variables, methods, modules and classes implemented but not used within the program, incorrect state models, incorrect messages, branching errors, algorithmic, and logical faults. They also highlighted that preliminary work on object-oriented systems suggested that the majority of faults occurred in the interface between objects and not intra-object.

Hayes [38] made an attempt at a taxonomy of object-oriented defect types by consolidating the defect types found in the literature. To do this, Hayes examined several sources of object-oriented defects [33], [76]. Each of these sources investigated object-oriented defects and put forward possible test methods that could be applied to find the defects.

Defect Classification

(A) Instance Variables
- initialisation - improper initialisation of class instance variables
- improper values - incorrect/invalid value assigned to instance variable moving system to incorrect state
- improper usage - instance variable used at an incorrect place

(B) Methods
- returns incorrect value
- faults with algorithm in method
- if / while / other conditional faults, etc. - faults with structure of conditional statements

(C) Relationship
- hierarchy - class incorrectly placed within hierarchy
- failures associated with inheritance, implementation, method overriding

(D) Message / Interfaces
- correct message to wrong object
- incorrect message to right object

Figure 3.2 – Defect Classification for object-oriented code based on literature review

From these sources of information, a list was drawn up of all possible defects suggested for object-oriented code. Taking into account any overlap between defect types, these were abstracted and narrowed down to a list of four groups of defect type. These four groups, along with illustrative sub-classifications are shown in Figure 3.2. The classification is an approximation only, and in many cases, defects can fall into one or more of the groups.

For the experiment, the defects created fitted into the derived categorisation (Figure 3.2). Some of the defects used were naturally occurring, i.e. were identified in the code during development by the course lecturer, the others were seeded in the code based on the information gathered from the literature review. Two sets of defects were prepared for the experiment to maximise the number of possible defects seeded. Ten defects were present in the code given to group A and ten defects were present in the (same) code given to group B. One defect was present in both groups, with three other defects being similar in nature, but the syntax varying between the groups. 30% of the defects seeded were general defects (based upon historical experience of the course lecturer) and the remaining 70% were related to object-oriented code characteristics. A full list of the defects present in group A and group B can be found in Appendix A.2.7 and Appendix A.2.8.

Data Collection

Data from the inspections was collected via a defect report form (an example can be found in Appendix A.2.1). When a subject found a defect in the code they would record the time it was discovered, its location in the code, and a textual description that accurately described the defect. The defect report form was tested during the initial training phase of the experiment.

Data Analysis

Since the goal for the experiment was exploratory in nature, the results were investigated through the analysis of the qualitative information gathered.

Threats To Experimental Validity

An empirical study can be distorted by influences that may affect the experimental variables without the knowledge of the researcher. This possibility should be minimised as much as possible. Possible threats to internal validity included:

- Selection effects that may occur through variations in the natural performance of individual subjects. As part of an earlier exercise in the class, the subjects had been split into 12 groups of roughly average ability. For the inspection exercise, groups 1 to 6 were then assigned to group A, and groups 7 to 12 were assigned to group B. This should have minimised much of the possible effect.
- Plagiarism was not a concern as the experiment was carried out under exam conditions.
- The learning curve for the subjects associated with the programming language used (in this case Java). Prior to this class subjects had previously used the object-oriented languages of Eiffel and C++. To reduce any possible effect due to the use of a new language, earlier sections of the class had the subjects (in groups of 3 or 4) code a small program (approximately 8 pages in length). This was followed later by a more substantial library system. The average number of classes created for this task was 21 (ranges of 13 - 40) with an average length of 2755 lines of code (ranges of 1200 - 4500). The ranges vary so much due to some groups implementing a full graphical interface. The code used for the inspection exercise proper was an extension to this system.
- There was no monitoring of subjects prior to the experiment while they worked in groups. Some of the subjects may have worked at different rates, taking on more, or less, responsibility. This could have lead to an imbalance of subjects' knowledge and understanding of the system and perhaps skewing some of the experiment results.

Threats to external validity limit the ability to generalise any results from an experiment to a wider population. These threats included:

- The subjects of the experiment (3rd year Computer Science students) may not be representative of the general software engineering population. This could not be avoided due to time and resource constraints.
- The Java code may not be representative (in complexity or stylistically) of industrial software. In this case, the code inspected was part of a substantially larger software system, diminishing some of the complexity arguments.
- The defects seeded in the code may not be representative of the problems currently experienced in industry. As was mentioned earlier, a thorough search of the literature was carried out, the results of which were used to base decisions on types of appropriate defects.
- The inspection process used during the experiment may not have been representative of industrial software practice. This experiment focused only on the individual defect detection phase and used the ad-hoc method as a baseline for code inspection. It did not involve any presentational overview by the author as the subjects were already familiar with the general system and the group collation phase was not relevant to the aims of the study.

3.2.4 Experimental Procedures

Training

In week one, an introductory lecture and training phase were carried out before the experiment proper. The lecture lasted approximately fifty minutes and introduced the basic premise behind inspections, their uses and problems. The training phase, which was carried out the day after the lecture, lasting approximately two hours, was run informally to allow subjects to ask questions and overcome any conceptual problems about the ad-hoc inspection process. The experiment proper was held one week after the training exercise.

Conducting the Experiment

In week two, the experiment proper was held. No lecture was given in week two. Subjects were given up to a maximum of two hours to complete the inspection. They were supplied with a booklet containing the inspection task material (code, specification, class diagram, defect report form). If subjects went beyond the two-hour limit for the inspection, they were asked to stop working. The inspection task was completed under exam conditions to ensure that subjects worked independently.

3.2.5 Experiment Results

As a result of attrition, group A was reduced from 23 to 18 subjects, and group B was reduced from 24 to 23 subjects. The experimental results are summarised in Table 3.1.

Group	A	B
No. of subjects	18	23
No. of defects in code	10	10
Average no. of defects found by subjects	6.28	6.65
Average time for inspection (min)	76.93	80.04

Table 3.1 – Summary of Inspection results

Figure 3.3 shows the mean rate of defect discovery by both groups. It shows that the performance of both groups, A and B, was similar. This suggests that the balance of defects for both groups was similar. What can also be seen is that beyond the 60-minute mark, there was an average of only 0.5 defects found per subject.

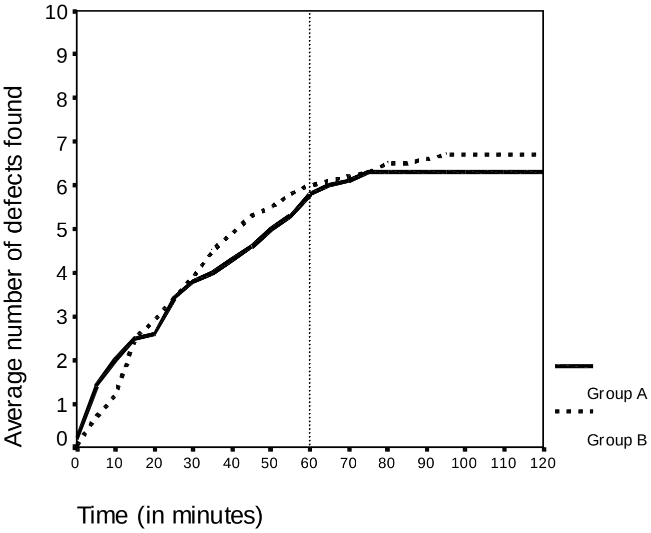


Figure 3.3 – Mean rate of defect discovery for groups A and B

Inspection Strategy

When subjects identified a defect in the code they noted the time at which it was discovered. This allowed a picture to be built up of the order and time at which defects were found. This timing information also provided an indication of how subjects carried out their inspection.

Figure 3.4 and Figure 3.5 show boxplots of the times the defects were found. Defects are listed in the order in which they appear in the code handed to the subjects.

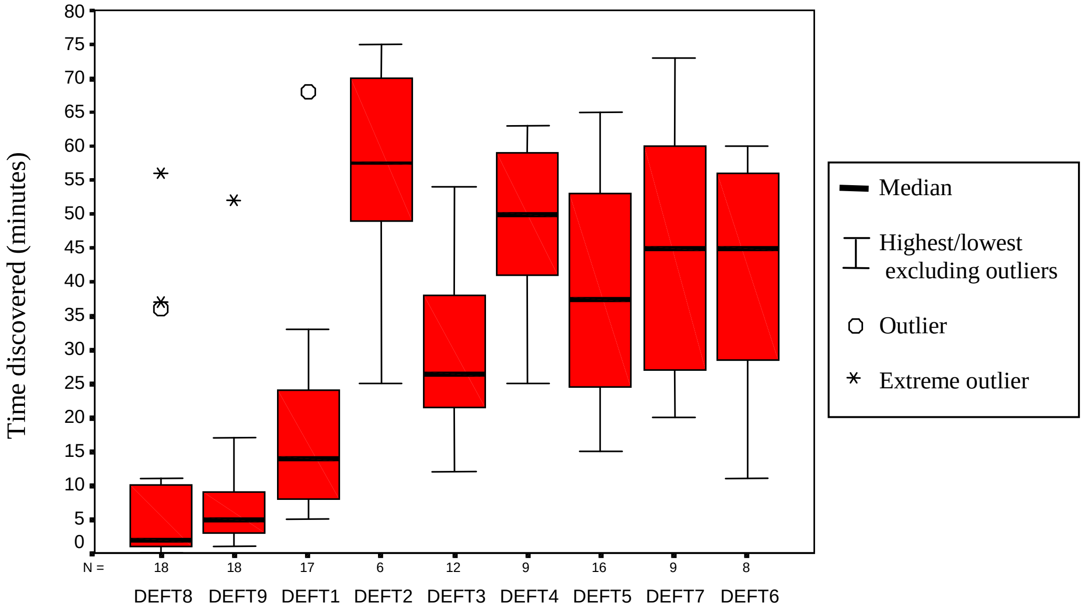


Figure 3.4 – Boxplot of defect discovery times for Group A

In general, the first three defects (defects 8, 9 and 1) in group A (Figure 3.4) were found in order. At least during the beginning of their inspection, subjects seemed to be reading through the code in the order it was provided.

The next defect that group A subjects should have found was defect 2. Five of the six subjects who managed to find this defect discovered it much later on. This particular defect involved the incorrect placing of a call to a method called `purge`. The call should have been several lines later in the code. In order to notice that the method call was misplaced, subjects had to gain a greater understanding of the code presented to them, including the role of the `purge` method, which appeared later in the code. This also suggests that subjects read through the code in order, rather than jumping to a method definition when it was called in the code.

Defect 4, the other defect discovered out of presentation order concerned an out of date reservation (the defect is discussed later in Section 3.4, and is shown in Figure 3.11).

For the remaining defects (3, 5, 7 and 6), their standard deviation (shown in Figure 3.4) is larger than that for the first few defects. Although it appears that, in the main, these defects were found in their presentation order, this cannot be stated with as much certainty. For both sets of defects, following defect 3 should have been defect 10, but it does not appear in either Figure 3.4 or Figure 3.5 because no subject discovered it. This was a particularly subtle defect involving iterating through a vector whilst deleting its elements.

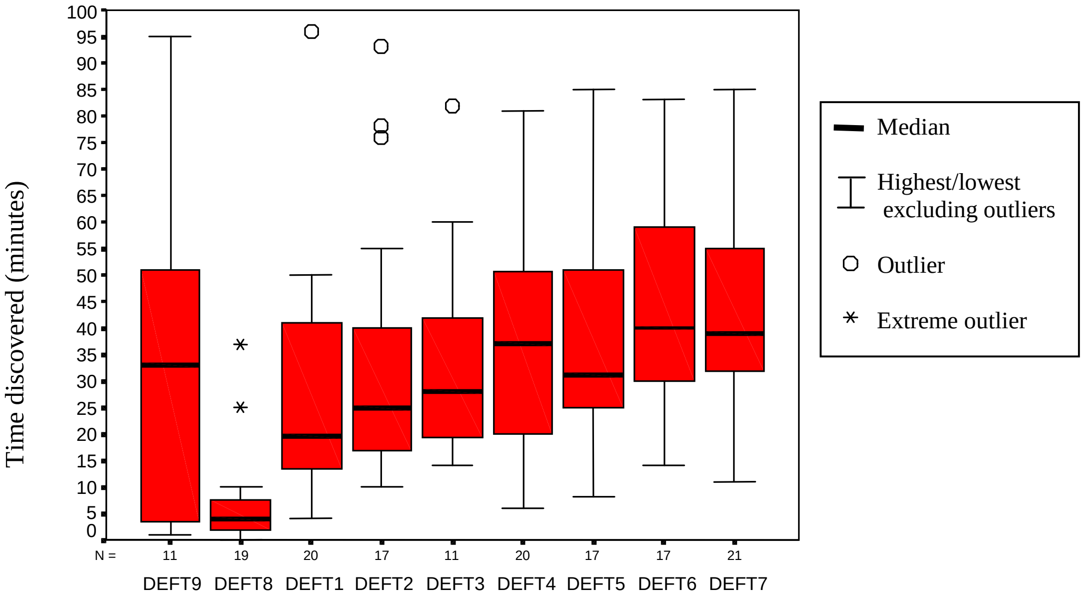


Figure 3.5 – Boxplot of defect discovery times for Group B

Looking at Figure 3.5, the first defect in group B's code handout was defect 9, but it was the first defect found by only four of the eleven subjects who correctly identified it. The others found this defect beyond the 30-minute mark, possibly suggesting that they found this defect during a second pass through the code. This defect involved one class, `Reservation`, implementing the `Enumeration` interface[1] (a predefined Java interface). The `Reservation` class erroneously provides the methods

[1] Java allows only single inheritance, but supports the use of interfaces. These specify a reference type, consisting of a type name and a set of abstract method declarations. A single class may implement many interfaces.

`hasMoreElements` and `nextElement`, which have to be defined by a class implementing the `Enumeration` interface.

Although there are a few outliers, Figure 3.5 suggests that group B also worked through the code and discovered defects sequentially in the order it was provided.

For the subjects that found the most defects (two subjects with 9/10 and seven subjects with 8/10) the average inspection time was 71 minutes. There was a very strong indication that these subjects read through the code in the order provided. Both of the subjects with 9/10 and one with 8/10 found all their defects in their presentation order and in their first pass through the code. The timing information from the remaining subjects suggests that they also read through the code in its presentation order, but that they required multiple passes through the code to find all their defects.

For the subjects that found the least number of defects (1 subject with 2/10, one with 3/10 and 5 with 4/10) the average inspection time was 89 minutes. It was difficult to see any pattern or strategy employed by these subjects with the information gathered, although there was some small indication that like their colleagues, they made multiple passes through the code in the order it was presented to them.

The results indicate that both groups of subjects read the code in sequential order - which would not be surprising given that they were new to inspection and were using an ad-hoc reading approach. This was also found to be true by Laitenberger *et al*. [52], who commented that "*without any guidance on what to check, most of the inspectors often perform their scrutiny sequentially. They start their checking activity at the beginning of the document and read through the document page after page* ". Those defects that were discovered out of presentation order (defects 2 and 4 in group A and defect 9 in group B) had a relatively low discovery rate. It may have been that defects that required more than sequential reading were harder to find. It may also have been the case that weaker subjects, by making more passes through the code, were finding the more difficult defects in later passes.

The Defects

To investigate possible links between defect characteristics and ease of detection, it was first necessary to group the defects. To do this all defects from groups A and B were brought together. This is presented in Figure 3.6, which shows the percentage of subjects (y-axis) who found each particular defect (x-axis) and which group the defect belonged to (the colour of the bar). This clearly shows which defects were discovered relatively easily and those that were harder to identify. To investigate whether there was any common factors between defects with similar detection rates, a series of characteristic words was compiled for each defect reflecting its key features (brief descriptions are shown in Figure 3.7). Figure 3.8 shows the keyword characteristics for the defects in percentage response order.

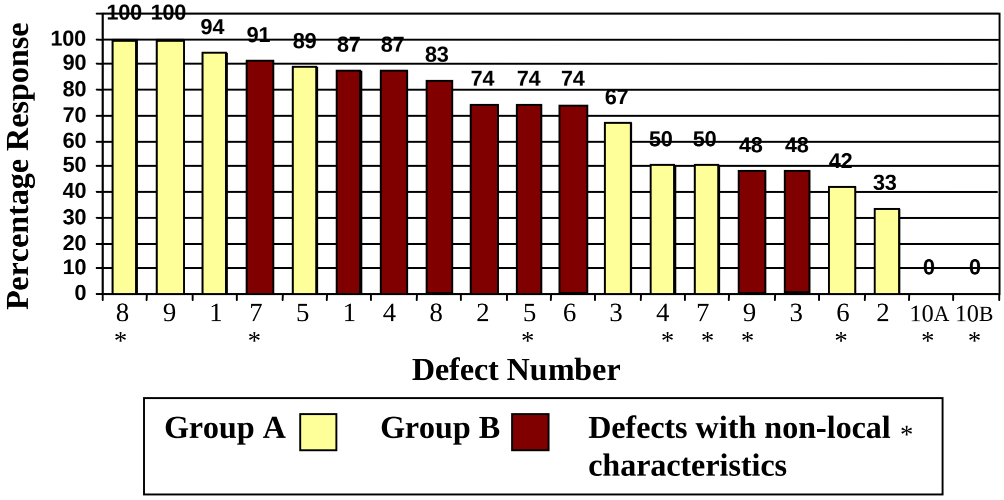


Figure 3.6 - Percentage of subjects finding each defect during the inspection

To investigate whether there was any similarity of characteristics between defects with similar percentage response rates, particularly the ‘hard to find’ defects, the information contained within in Figure 3.8 was entered into C5.0 [16], a data-mining tool. A data-mining tool attempts to draw out patterns from a set of provided data, allowing an objective view to be taken of the data set, free of any human preconceptions. The tool also allows experimentation with different groupings of data, to see how the generated patterns are affected. An example of its use can be found in Appendix A.4 and all the output generated from C5.0 can be found in Appendix A.5. From information generated by C5.0, and the information contained in Figure 3.8 the following points were observed:

- Locality of defects was well mixed, but harder to find defects tended to have class or system locality (see Figure 3.7 for definition of these terms).
- Defects involving class libraries and wrong messages tended to be harder to find.
- Method sizes were mixed but no harder to find defects appeared in small methods.
- Defects involving inheritance, overriding and design mismatches tended to be hard to find unless there was supporting domain knowledge.

- Defects involving a domain knowledge clash or instance variables had a very high probability of being found.
- Defects which had no domain knowledge clash but had diagram conflicts (i.e. involved inheritance, overriding, abstract classes etc.) had a less than 50% chance of being found.

Locality - area of code required to be looked at to identify the defect

(M)ethod -	information required to identify defect is present in one method
(C)lass -	information required to identify defect is present in one class
(S)ystem -	information required to identify defect is distributed across multiple classes

Algorithm/computation - defect due to an error in the algorithm

Use of library class - defect requires understanding of class libraries

Wrong object - defect caused by sending message to wrong object

Wrong message - defect caused by sending incorrect message

Data flow error - defect caused by lack of variable usage or variable mis-usage (variable used in incorrect way)

Method size - size of method where defect present

S	= 0-4 lines of code
M	= 5-10 lines of code
L	= 11+ lines of code
-	= defect does not reside within a method (e.g. class definition or missing method)

Instance variable misuse - defect due to assigning incorrect values to instance variables

Omission - defect associated with missing code

Commission - defect associated with incorrect or superfluous code

Inheritance/implementation - defect associated with inheritance/implementation

Override - defect associated with method overriding

Diagram mismatch - defect associated with inconsistency between code and documentation

Domain knowledge - defect associated with a clash between subject's knowledge of the domain and the code

Figure 3.7 – Description of defect features

In Figure 3.6, all of the defects with '*'s at the bottom of their columns all contain *non-local* characteristics. Information outside the 200 lines of code being inspected, but still available to them, was needed for a full understanding of the defect. Defects 8A and 7B were considered easy to find as they involved diagram mismatches and clashed with inspectors' domain knowledge (shown in Figure 3.8), and defect 5B, although requiring some non-local information could be guessed by making reasonable assumptions about the method where the defect resided. The remaining defects with non-local characteristics had a discovery rate of 50% or less.

Table Notes

Locality:
(M)ethod
(C)lass
(S)ystem

Method size:
(S) 0-4
(M) 5-10
(L) 11+
(-) No size available

	8A	9A	1A	7B	5A	1B	4B	8B	2B	5B	6B	3A	4A	7A	9B	3B	6A	2A	10A	10B
Locality (M, C, S)	S	M	M	S	S	M	S	M	M	M	M	M	S	S	S	M	S	C	S	S
Algorithm/computation		X				X				X				X				X	X	X
Use of class library													X	X					X	X
Wrong object											X									
Wrong message												X	X	X		X				
Data flow error			X					X	X		X	X				X				
Method size (S, M, L)	-	M	S	S	M	L	M	S	L	M	M	L	M	M	-	M	-	L	M	M
Instance variable misuse					X		X	X												
Omission			X						X				X				X			
Commission	X	X		X	X	X	X	X		X	X	X		X	X	X		X	X	X
Inheritance/implementatio	X														X					
Override				X													X			
Diagram mismatch	X			X											X		X			
Domain knowledge	X			X																
%	100	100	94	91	89	87	87	83	74	74	74	67	50	50	48	48	42	33	0	0

Figure 3.8 - Defects described by their features

Amongst the harder to find defects (i.e. found by less than 50% of subjects) there were only two that had local characteristics, defects 3B and 2A. Defect 3B involved a data flow error (a variable was passed as a parameter then never used), but was completely local to the method. Defect 2A was an algorithmic error, with the misplacing of a method call in a series of `if-else` statements. There are no similar characteristics between these two defects.

3.2.6 Experimental Design Lessons

This section highlights some of the lessons learned from running the experiment and suggests ways in which it could have been improved.

Subjects used the defect report forms to record when they found each defect. This allowed the order and timing of defect discovery to be investigated. However, it was impossible to be able to accurately describe the order in which subjects read through the code. Only suggestions could be made, based upon the defect discovery times. To build up a more complete picture of subjects reading strategy, more timing information must be gathered, e.g. gathering the time subjects begin to read a method.

During the inspection, subjects were also allowed to use some form of Java reference material, e.g. a Java code book, the on-line reference guide for JBuilder (a Java programming environment), or the on-line reference guide provided with Java for its class library. This material may have helped subjects find some of the seeded defects. During the experiment, subjects were not monitored on a one-on-one basis due to the large number of participants. Because of this, no comments could be made on the usefulness of the reference material available.

One way to help gather this kind of information would be the use of verbal protocols. A subject is monitored through an entire experiment and verbally states what he/she is thinking and doing. All this is recorded (via either audio or video equipment or a supervisor taking notes) and properly reflects the cognitive process employed by subjects during an experiment. There are however some down sides to using verbal protocols. They require at least one supervisor per subject. This is because in most cases the supervisor has to prompt the subject, gently reminding them that they are supposed to be verbalising their thoughts. Also, because a supervisor is required for each subject, this limits the number of subjects that may be recorded. A cheaper solution to this problem may be to use questionnaires given to subjects after the completion of the experiment.

3.2.7 Summary

Through the creation of a key word classification index for each defect and the utilisation of a rule induction system, the experimental results suggest a major contributor to difficult to discover defects is that information required to understand the defects is not available locally. Instead, the information is distributed throughout the code by the features of the object-oriented language. The study of reading strategies used by subjects found that most appeared to read the code sequentially. Object-oriented code is not naturally sequential. It is unclear if this disparity may affect the detection of defects with non-local characteristics.

3.3 Survey of Object-Oriented Defect-Detection Approaches

The experiment described in the previous section suggested that several object-oriented features such as message passing, class libraries, interfaces and method overriding could make defects difficult to detect. To further investigate the problems object-orientation can cause for code inspection and to obtain confirmatory evidence for the previous experiment (following the multi-method approach suggested by Daly [21]), a survey was created to obtain the opinions professionals in industry.

3.3.1 Survey Design

The following describes the creation of the survey, stating the initial objectives, justifying the chosen method of delivery, style, and layout of the survey, as well as the questions to be included.

Objectives

The objectives of the survey were:

- Investigate current practices in industry concerning removal of defects from object-oriented code.
- Find out if inspections are being carried out on object-oriented code, and if so, how.
- Investigate further some of the key findings from the experiment described in the previous section.
- Gain a better understanding of the features of defects that are causing problems for object-oriented software developers.

Survey Method

Various methods can be used to gather information. These include personal interviews, telephone interviews, postal surveys, etc. Due to limitations on the time available, locality of businesses, and availability of willing participants, it was decided that interviews would not be practical, and that some form of survey would be more suitable.

It was further decided that the surveys used to elicit the desired information would be sent via email rather than by post. The reasons for using email to deliver the surveys include:

- Cost - no paper or envelopes are involved.
- Response time - since the information is passed electronically, delays should be kept to a minimal.
- Extra space - unlike paper based surveys, making extra space for answers in an electronic survey is not a problem.

There are however several disadvantages to using surveys to gather information. These include:

- Follow on questions - unlike in an interview, you cannot ask a respondent to refine their answer, or to probe further. See the next section on survey construction for more on this.
- Low response rates - there is a danger when sending out surveys that you get a low response rate. For mail based questionnaires, Edwards [29] found a response rate of 20-30%, and in a software maintenance study, Lientz and Swanson [58] only received a 24.6% response rate. In an attempt to reduce this problem various companies were contacted prior to the completion of the survey, via University of Strathclyde graduates now in their employment, to see if they would be interested in participating.

Survey Construction

The time required to complete the survey was chosen to be approximately thirty minutes. Anything over thirty minutes and it was judged that respondents might have been less motivated to reply.

Oppenheims' template for survey construction [68] was used as the basis for this survey. Oppenheims' template consists of several sections, (1) a prologue used to inform the respondent of the topic, (2) a classification section used to obtain personal details about the respondent, (3) an information section containing the questions on the topic under consideration, and finally (4) a closing section thanking respondents for participating and providing instructions for returning the survey. In this survey, the classification section enquires about the respondent's history and experience with object-oriented languages; personal details like name and age were not required, and instructions on how to return the survey were included in the introduction section, as well as the closing section.

As indicated previously, a survey was being carried out, not an interview. Probing respondents on a response or asking them to clarify a point is not usually carried out with surveys, especially if the respondents are anonymous. For this reason, an extra question was added to the closing section asking respondents if they wouldn't mind answering one or two follow-on questions if the need arose, and to include their email address.

The main body of the survey, the information section, contained the core questions. This section was further split into two sub-sections; the first contained questions dealing with methods of defect detection and further asked the respondent if they carried out code inspections, the second contained questions dealing with defects and their characteristics.

When writing the questions for the survey, there were many points that had to be taken into account. Questions should only ask for one piece of information, question wording should not imply a desired answer, question wording should not have a double meaning, and should not use abbreviations that may not be understood [12]. Sinclair [86] stated that questions should be understandable and unambiguous, and that they should be as short as possible. Other important points on the art of question construction for surveys can be found in [11], [34].

Survey Questions

For the purposes of the survey, the classification section was used to detail the respondent’s current job position and duties, as well as the object-oriented languages encountered and what their roles have been in relation to object-oriented software, e.g. programmer, designer, or tester. This information was used to highlight respondents’ experience with object-oriented software as well as the languages used within industry.

The second section dealt with the process of defect detection. It was important to get an idea of the current processes used within industry to remove defects from code. The remainder of the second section was dedicated to questions on one specific defect removal technique, code inspection.

Based upon the experiment described in Chapter 3.2, Roper and Dunsmore [81] suggested several aids that could be used to help inspectors with the more awkward/hard to find defects in object-oriented code. These aids included:

- Checklists - a series of questions that guide programmers to aspects of code that have a high probability of containing defects.
- Perspective based reading - multiple inspectors, each using a different perspective, e.g. tester, designer.
- Visualisation - can be as simple as modifying the size, colour and style of code, or involve diagrams showing relationship between classes, objects, method calls, etc.
- Contextual access - the use of hypertext links (commonly used in web pages) to access related information from the code under inspection.
- Experience base - a database of lessons learned from defect detection techniques, defect models, as well as project specific lessons.

Several questions relating to these aids were placed in the survey to obtain the views of professional software engineers, and to gauge whether, in their opinion, any of these aids were worth further investigation.

The third and final section investigated object-oriented defects and their characteristics. The first few questions were designed to elicit the knowledge of the respondent on what they believed were characteristics of hard to find defects, and their views on what helped them find these types of defects. The remainder of this section presented respondents with a list of characteristics, which could be associated with defects. This list was derived from the characteristics of the defects used in the experiment described in Chapter 3.2. This would allow a comparison between the characteristics of problematic defects in that experiment and the defect characteristics found to cause problems for professionals in industry.

Once the survey was completed, a trial run was carried out using several postgraduate students and lecturers from the Computer Science Department at University of Strathclyde. From the trial run, several questions were reworded and modified. These initial results were not included in the final analysis of the survey. Once the modifications were completed, the survey was sent via email to those industrial contacts that had expressed an interest in participating. The survey was also posted to two of the main newsgroups for software engineering (*comp.software-eng* and *comp.software.testing*). A full copy of the survey can be found in Appendix B. The next section summaries the responses received.

3.3.2 Survey Results

Object-Oriented Background

Thirteen responses were obtained, at least half from senior software engineers or managers. Although thirteen responses are insufficient to draw significant conclusions, there are enough responses to obtain an insight into the current state of industrial practice. The average length of time respondents had been working with object-oriented code was 4.8 years (ranging from 1 to 12 years). Most respondents had participated in all aspects of the software development process. All had used C++, with four having used Java. Table 3.2 shows a list of all the object-oriented languages used by respondents.

Language	**Number of respondents**
Ada95	4
C++	13
Eiffel	2
Forte (Transaction Object-Oriented Language)	1
G2	1
Java	4
Modula-3	1
Object Pascal	1
Objective C	1
Perl	2
Python	1
Visual Basic	1

Table 3.2 - Object-oriented languages used by survey respondents

Defect Detection

Respondents were then asked what approaches they had used to detect defects in software and at what particular stages in the software lifecycle. Replies showed a multitude of techniques used over many parts of the software lifecycle. A summary of the replies are shown in Figure 3.9.

1. Analysis/Design
Fagan reviews, Checklists, Databases, Walkthroughs, Individual & Team Reviews, Requirements review, System design review
2. Code
Reviews, Fagan Inspections, Debuggers, Memory debuggers, Complexity analysis tools, other commercial tools, e.g. Pro Lint
3. Testing
Unit testing, Integration testing, Acceptance testing, System testing, Dedicated test applications, Alternate compilers, Functionality testing, Coverage testing, Commercial tools (e.g. Purify), Peer testing, Custom/generic testing set-ups, Site testing

Figure 3.9 - Defect detection approaches used in industry

From the thirteen that replied, twelve had carried out a code inspection or review of object-oriented code. Table 3.3 summarises the aids used to help carry out object-oriented code inspection. In some cases respondents stated more than one aid. The most popular inspection aid was the checklist, followed by Perspective Based Reading. Three respondents used no aids at all.

Aids for inspection	**Number of respondents**
Checklists	5
Perspective Based Reading	2
References to Design Documents and Requirements	1
Compiled Code	1
Tool (Prolint)	1
Code Walkthroughs	1
None	3
No Answer	2

Table 3.3 - Aids used to carry out object-oriented code inspections

When carrying out code inspections, respondents used several reading strategies; sequential (7), top down (4), bottom up (1), class wise (1), and two subjects stated that they used no particular reading strategy. In some cases, respondents used more than one reading strategy.

The amount of code inspected and the time taken to inspect varied enormously. For half the respondents, inspections usually took no more than two hours. In those two hours anywhere between 50 to 1000+ lines of code were inspected, where the code included full classes, important methods, functional units, etc. In several cases, inspections were carried out sporadically and in bulk, leaving half or a full day to do all the inspections for several weeks work.

Respondents were presented with five techniques that could be used to help with object-oriented code inspection and were asked to indicate the extent to which they thought those techniques would be beneficial (ranging from strongly disagree to strongly agree). Table 3.4 shows the results for the twelve respondents who carry out object-oriented code inspections. Although in general most seem to believe that all the suggested methods can be useful, visualisation appears to have the strongest support, followed by checklists. A brief description of each of the techniques shown below can be found in Question 12 of Appendix B.

	Grading				
Technique	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
Checklists	0	0	1	6	5
Perspective Based Reading	0	1	2	6	3
Visualisation	0	0	1	5	6
Contextual access	0	1	4	5	2
Experience base	0	0	3	5	4

Table 3.4 - Possible aids for object-oriented inspection

Following on from the previous question, respondents were asked for any views or comments on any of the five techniques. Figure 3.10 summarises their responses.

Checklists
• Eventually are built into development procedure – more effective as prevention than detection • Useful for novice programmers
Perspective based reading
• Used for requirements and design • Too expensive for coding • Always use some form of perspectives in code review
Visualisation
• Useful in code reviews • A risk is that the picture is the subject of the review/inspection rather than the code
Contextual Access
• Would require extensive case tool and not guaranteed to help find more defects
Experience base
• Could become unwieldy and difficult to find information in • Useful, although this is people dependent rather than technology dependent
Other comments
• Easier to review on paper • Any additional information which helps with understanding is useful

Figure 3.10 - Comments on possible aids for object-oriented inspection

Defects

The survey asked respondents to list any object-oriented features that were a common cause of defects or created difficulties in understanding code, as well as to list any techniques/tools that may be useful in those circumstances. The responses are summarised in Table 3.5 (first row problems; second possible aids, if any). Two respondents listed inheritance and another highlighted multiple inheritance as features liable to cause defects. Other features listed by respondents relating to difficulties in understanding object-oriented code include multitude of dependencies, deep hierarchies, and dynamic binding.

Respondents were then presented with a list of characteristics that could be used to describe a defect and asked to select those that they felt caused the most problems. These were based on characteristics identified in the experiment described in Chapter 3.2. Table 3.6 shows the number of survey respondents who highlighted each particular defect characteristic.

1	Call dispatches, fulfilling requirements (operation doing too much or too little)
	Comments in code
2	Multiplicity of connections making analysis difficult
	Simple public interfaces, classes should be as stand alone as possible
3	Flow of execution, deep hierarchies, use of exceptions
	Debugger (allowing line by line step through), good documentation
4	Dynamic binding, difficulty in following flow of control
	-
5	Inheritance
	Tags in Emacs
6	Multiple inheritance, location of variable definitions, i.e. which class?
	-
7	Inheritance, complicated objects (records)
	Reviews on design (models) and code reviews
8	Runtime control flow fragmentation across small functions, use of simple methods does not make methods themselves simple, complexity resurfaces in interactions between objects, sometimes not clear what code is executing given a particular set of inputs
	Browser tools to allow definitions and references to methods, and variables help navigate round the code
9	Incorrect class modelling, complexity of C++
	More time spent on design, C++ best mastered with experience

Table 3.5 - Object-oriented features that were a common cause of defects (first rows) and potential aids (second rows)

Twelve of the thirteen respondents reported that problematic defects had information required to identify them distributed across multiple classes, and ten respondents claimed that features of inheritance could cause problems. Several respondents also claimed that control flow was a problem when attempting to understand object-oriented code, especially due to the presence of many small methods. Other high responses were for problems with the algorithm, inconsistency between the documentation and code, variable misuse, and missing code. When asked if there were any other characteristics that may cause problems for object-oriented code, documentation was a recurring problem, whether it be documentation of the code under inspection or code belonging to third party libraries.

3.3.3 Summary

From the information gathered it appears that a variety of different techniques are used at different stages in the software lifecycle in an attempt to remove defects, from the original design documentation, right through to testing and deployment. Twelve of the thirteen respondents had carried out object-oriented code inspections, most of which seem to be using the same ideas and aids that have been used previously for procedural code inspections.

Characteristics	**Number of respondents**
All information required to identify a defect is distributed across multiple classes	12
Defect associated with inheritance	10
Defect is due to an error in an algorithm	7
Defect associated with inconsistency between code and documentation	7
Defect caused by variable misuse (data flow)	6
Defect associated with missing code	6
Defect associated with a conflict between requirements and code	5
Defect requires understanding of class libraries	4
Defect is in a method of size greater than 10 lines	3
Defect caused by sending message to wrong object	2
Defect caused by sending incorrect message	1
All information required to identify a defect is present in one class	1
All information required to identify a defect is present in one method	0
Defect is in a method of size less than 5 lines	0
Defect is in a method of size 5 - 10 lines	0

Table 3.6 - Characteristics of problem defects

Evidence from the survey appears to suggest that non-local information can be a real problem. Twelve of the thirteen respondents reported that problematic defects had information required to identify them distributed across multiple classes, and ten respondents claimed that features of inheritance could also cause problems. Several respondents also claimed that control flow was a problem when attempting to understanding object-oriented code, especially due to the presence of many small methods.

Although this was a small-scale survey, its use of professionals from industry provides complementary evidence for the literature and further supports the experimental findings presented in Section 3.2 that non-local information within object-oriented code causes many of the current problems for object-oriented code inspection.

3.4 The Problem of Delocalisation

The results of both the experiment and survey suggest that a major feature of difficult to discover defects is that the information necessary to understand the defect is not in one location but instead distributed throughout the code. Understanding a piece of code requires following a trail of method invocations through many classes, moving up and down the inheritance hierarchy [97] (see Figure 2.4). The evidence gathered from both the experiment and the industrial survey suggest that many of the more difficult to discover defects in object-oriented code contain this characteristic.

Soloway *et al.* [88] first observed this in the context of program comprehension. They described a 'delocalised plan[2]' as "*where the code for one conceptualised plan is distributed non-contiguously in a program"*. Soloway goes on to say "*Such code is hard to understand. Since only fragments of the plan are seen at a time by a reader, the reader makes inferences based only on what is locally apparent - and these inferences are quite error prone*". Wilde and Huitt [97] argue that features such as inheritance, polymorphism, and dynamic binding are major contributors to the manifestation of delocalised information within object-oriented code. To illustrate the concept of delocalised information with respect to defect detection, Figure 3.11 shows a piece of Java code that was part of the library system for reserving a video used in the experiment in Chapter 3.2.

```
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;
    }
  }
}
```

Figure 3.11 - Defect with delocalised information

[2] Letovsky and Soloway [56] "*use the term goal to denote intentions and the term plan to denote techniques for realising intentions*". A plan can be thought of as a related set of actions that together achieve a programming goal.

In Figure 3.11, the purge method checks elements of the reservations vector to see if any reservation has become out of date. The GregorianCalendar method after should have been passed a date (taken from the reservation currently being referenced in the reservations vector). Instead, the reservation itself was passed to the after method. The argument was missing the part which retrieved the reservation date from the currently selected reservation, **elementAt(i)).getDate()))**.

The code had been compiled with the Java compiler, so there were no syntax errors and the code could be executed. The Reservation class belonged to part of the library system under development, whereas the Vector and GregorianCalendar classes belonged to the Java class library.

To be able to fully appreciate this defect, a variety of sources of delocalised information have to be examined:

- Vector -Vector method elementAt() was used to retrieve an element from a specific location in the collection of reservations.
- GregorianCalendar - GregorianCalendar method after was used to compare two dates. The reason the code compiled was that the method accepted anything of type Object - the base class in the Java class hierarchy (all classes derive from the Object class). So in this case, the method after accepted an object of type Reservation because a Reservation object was, through inheritance, also of type Object.
- Calendar – GregorianCalendar is a subclass of Calendar and inherits much of its (quite complex) functionality as a result.
- Reservation class - was used in the Purge method to cast the object removed from the vector reservations. The missing method call should have been made to the Reservation method getDate().

As well as examining other classes in the system and the classes from the Java class library, system documentation is another source of information (illustrated in Figure 3.12). All of this information has to be understood before the defect can be correctly identified, and none of it is available locally within the method and class under inspection. This situation is by no means unusual, as object-oriented programming is based around such message passing and the use of other classes. This kind of delocalisation has been reported as causing problems for software maintenance [80], [98], and testing [46].

It should be noted that the problem of delocalisation also exists in a well-modularised procedural system, but that the key features of object-oriented languages make this problem much more pronounced.

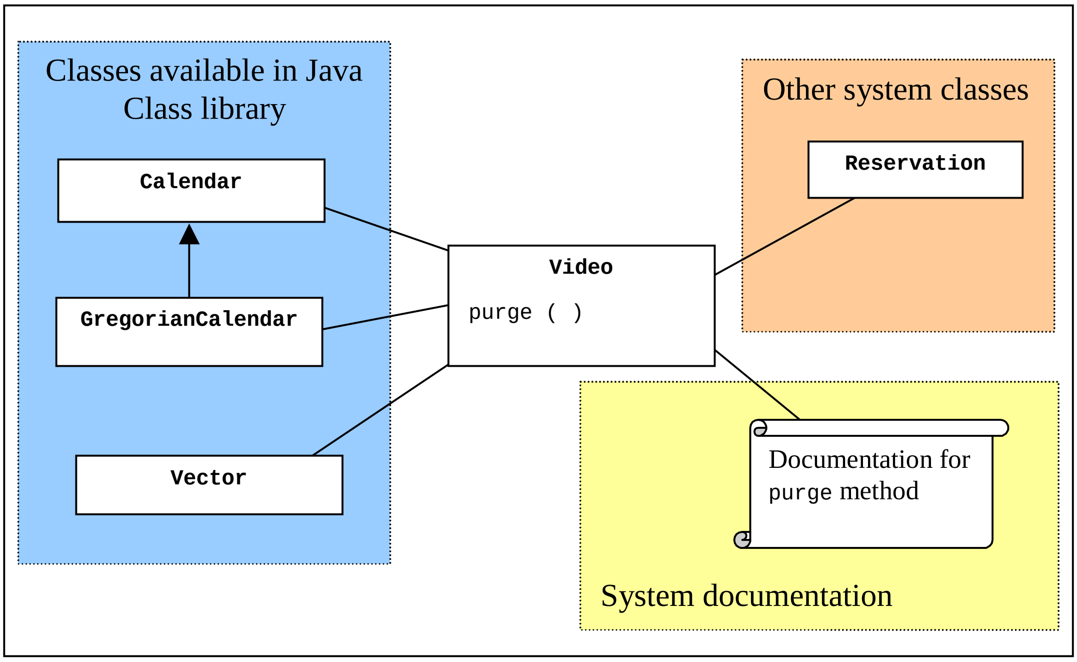
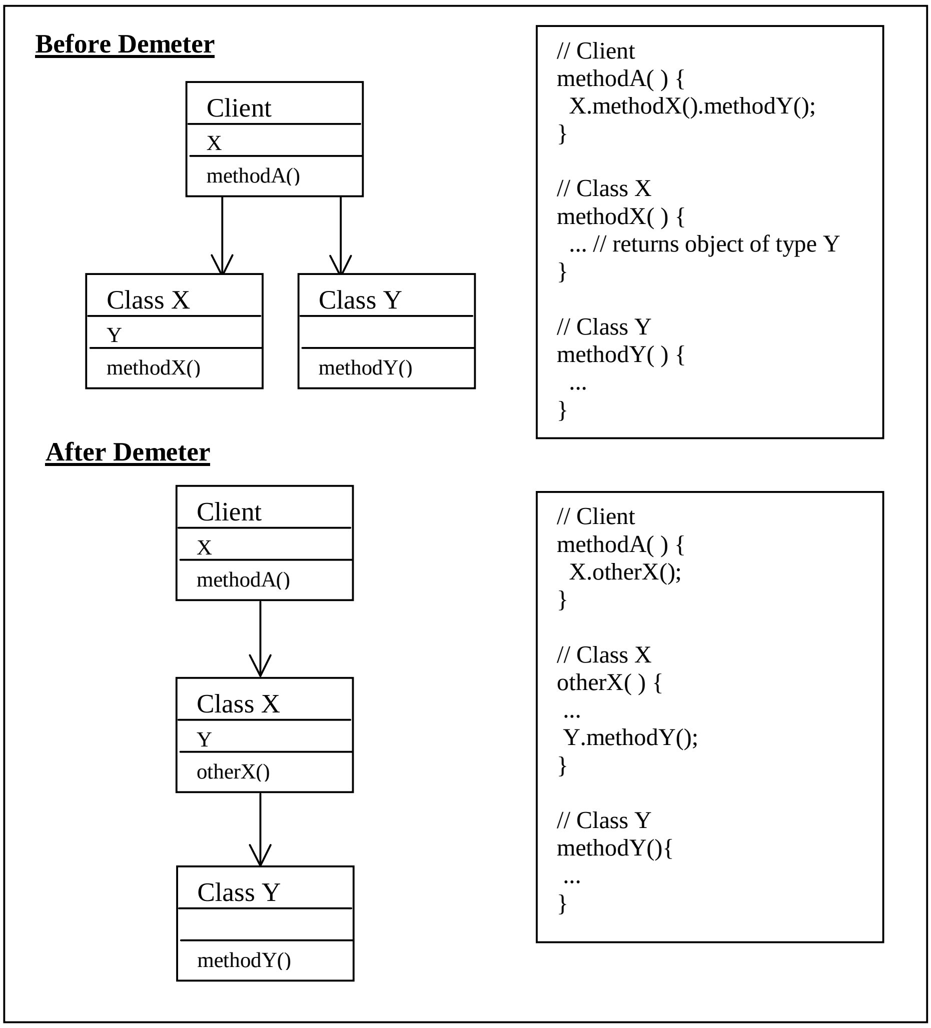


Figure 3.12 – Highlighting issues of delocalisation

In discussing good style for object-oriented programs, Lieberherr and Holland [57] presented the 'Law of Demeter'. The goal of the law is to restrict the message sending structure of methods (e.g. nested message sending), therefore reducing the number of dependencies between classes. It does this by restricting messages to 'neighbours', where neighbours are instance variables, method parameters, global variables, or objects created directly via a constructor. Although this reduces the delocalisation to immediate neighbours, it does not reduce the functionality that has to be looked at in order to understand what the code is doing. This is shown in Figure 3.13, where before applying Demeter, *methodA* in the Client class calls *methodX* on its instance variable X, which returns an object of type Y, which then accepts a call to *methodY*. After applying Demeter, *methodA* in the Client class calls *otherX* on its instance variable X, and method *otherX* in class X calls *methodY* on its instance variable Y. Although applying Demeter reduces Clients dependencies to class X, the same functionality is still present, only moved to method *otherX* in class X.

Other problems that can arise from following the law include an increase in the number of methods, an increase in the number of arguments passed to methods, and a decrease in code readability [57]. Potential advantages of using the law are that code can become easier to understand and maintain [57].



```
// Client
methodA( ) {
  X.methodX().methodY();
}

// Class X
methodX( ) {
  ... // returns object of type Y
}

// Class Y
methodY( ) {
  ...
}
```

```
// Client
methodA( ) {
  X.otherX();
}

// Class X
otherX( ) {
 ...
 Y.methodY();
}

// Class Y
methodY(){
 ...
}
```

Figure 3.13 – A dependency graph and code showing the outcome of applying the Law of Demeter

3.5 Inadequacy of Current Inspection Approaches

There are various reading techniques available that can be used by individual inspectors during the inspection of code. How much do currently available reading techniques help inspectors deal with the issue of delocalisation? Two of the oldest reading techniques are ad-hoc and the checklist (still regularly used), with Perspective Based Reading (PBR) a relatively new technique.

Consider the example shown in Figure 3.14. There are several flaws in the structure of the `cancelReservation` method. Given a person and a date, the method was supposed to remove the associated reservation that had previously been made for a particular video. The flaws in the method were:

```
public void cancelReservation(Person u, GregorianCalendar d)
{
  Reservation r = new Reservation(u,d);
  for(int i=0; i<reservations.size(); i++)
  {
    if (reservations.removeElement(r))
      System.out.println("Reservation Cancelled");
    else
      System.out.println("Cancellation Failed");
  }
}
```

Figure 3.14 – Example of a delocalised defect

- The use of a `for` loop when none was required.
- An assumption placed on comparisons made between the date held in `d` and the dates held in the `reservations` vector. The specification for the method stated that only the year, month and day were to be taken into account when comparing dates. However, the Vector method `removeElement` compares two objects using their equals method, meaning that in this case, the hour, minute and seconds values in both these objects were also considered in the comparison.

The ad-hoc reading technique offers no support to the inspector, instead the inspector relies on their own knowledge and experience and reads the code in their own preferred way. It offers no guide to help focus an inspector on parts of the code or help them comprehend the code. Whether any of the delocalised information is traced depends solely on what the inspector does. This suggests that ad-hoc reading may have poor results when dealing with delocalised information, and depends heavily on the individual inspectors. It certainly provides no active support to address delocalisation.

Checklists offer more support than ad-hoc in the form of a series of questions (see Figure 3.15 - from [40]), which must be answered by the inspector. One drawback of using a checklist is that it "*provides little support to help an inspector understand the inspected artifact*" [50]. It is unlikely that a checklist would highlight incorrect use of the date storage class `GregorianCalendar` in Figure 3.14 as the code is, in itself, functionally correct but contains the hidden assumption relating to the number of fields used in the date comparison. Although Porter *et al*. [70] commented that checklists might be thought of as systematic because they define reviewers responsibilities and ways to identify faults, they argue that the generality of the questions and lack of concrete strategies for answering the questions makes checklists a non-systematic reading strategy. Checklists do not encourage inspectors to follow the trail of delocalisation, they encourage *localised, as-needed reading* (see following section).

Complete	Verify that the code covers all the design.
Strings	Check that all strings are • identified by pointers and • terminated in NULL
{} Pairs	Ensure the {} are proper and matched
Calls	Check function call formats: • Pointers • Parameters • Use of ' &'
File Open and Close	Verify that all files are • Properly declared, • opened and, • closed

Figure 3.15 – Typical checklist questions, [40]

PBR, the newest code reading technique, has the goal to "*examine the various descriptions of a software artifact from the perspectives of the artifact's various stakeholders for the purpose of identifying flaws.*" [48]. Each inspector is given one perspective, each of which is different from the rest of the inspectors on the team. Examples of perspectives include designer, tester, and maintainer. Multiple inspectors are required to obtain a ‘reasonable’ level of coverage of the document. Each perspective contains instructions on extracting the relevant information for examination (in respect to their perspective), and is followed by a series of questions to be answered based on the information collected. In this way, PBR encourages a better understanding of the code but, like checklists, it doesn’t actively encourage inspectors to follow the delocalisation trail.

Another weakness of all three approaches – ad-hoc, checklist and PBR – is that none of them help reduce the amount of code that would have to be understood if delocalisation trails were followed. Following the trails is necessary for a sufficient understanding of the code to help identify delocalised defects. An inspection on 200 lines of object-oriented code could easily swell by an order of magnitude due to inter-class dependencies. All of the approaches assume that a manageable quantity of code (e.g. 100 lines per hour) can be easily isolated.

Thus the reading techniques ad-hoc, checklist and PBR are not designed to cope with defects where the information required to understand and identify them is delocalised. They neither encourage inspectors to follow the trails of delocalisation nor help reduce the amount of code to be read if the delocalisation trail is followed. New techniques and aids are needed to address these problems.

3.6 Ways to Improve Object-Oriented Inspection

There are two general strategies that can be used when trying to understand program code. These are systematic and as-needed understanding that Soloway *et al.* [88] described in the context of comprehending a program for the purpose of maintenance:

Systematic Strategy: *Programmers using this strategy started at the beginning of the program and documentation and traced the flow of the entire program, using various forms of simulation (e.g. symbolic, actually plugging in values)* [88].

As-needed Strategy: *Programmers using this strategy chose to study portions of the code and documentation, which they believed would be useful for constructing their enhancement. They read those portions as they decided that they needed them* [88].

The problem of delocalisation means that the information required to spot defects can be spread over many classes, methods and libraries, and can even involve dependencies on code that hasn't yet been written. This creates the effect shown in Figure 2.4, that to fully understand what one method is doing, a string of method invocations have to be followed, perhaps up and down a class hierarchy, and dramatically exploding the amount of code that has to be looked at.

Systematically inspecting (and understanding) all code and its dependencies would provide the understanding required to identify delocalised defects. However, due to the size of real systems, this would be expensive and time consuming. Also, due to limitations on the amount of information that can be usefully retained at one time in short-term memory, it would be unrealistic for one person to understand an entire system. More practically, when inspecting object-oriented code, an as-needed reading approach has to be adopted to deal with the possibly large amounts of delocalised information. This would allow inspectors to select the parts of the system they believed were necessary to develop their understanding. However, the danger is that an as-needed approach will force inspectors to make unverified assumptions that lead to the kinds of defects illustrated in the previous experiment being missed.

A related problem is how to select the code to be inspected. Due to the large number of dependencies within object-oriented code, it becomes very difficult to isolate an appropriately sized chunk of code. Selecting by size alone is inappropriate due the many links and dependencies one class may have. The aim must be to limit these dependencies.

For inspections to be effective for object-oriented code, techniques and aids need to be developed that specifically address delocalisation. In particular the following issues must addressed:

(1) Chunking - The many dependencies and links between classes make it very difficult to isolate even one or two classes for inspection, and delocalisation complicates this further. How you partition the code for inspection defines what an inspector gets to inspect. Two issues in this respect need to be addressed: (1) the identification of suitable chunks of code to inspect, and (2) decide how to break the chunk free of the rest of the system, minimising the number of dependencies and the amount of delocalisation.

(2) Reading Strategy - How should object-oriented code be read, especially if systematic reading of code is impractical? Is there a reading strategy that could help inspectors deal with delocalisation? Can checklists or PBR be modified to address delocalisation or are new reading strategies required?

(3) Localising the delocalisation - A way has to be found to abstract the delocalised information for the inspector, providing the benefits of systematic reading without the unrealistic requirement that *everything* is read.

3.7 Conclusions

The chapter has presented a consistent body of evidence using existing literature, an inspection experiment, and a small-scale survey of industrial practice that suggests delocalisation is a significant problem for the application of traditional inspection techniques to object-oriented code. Well-structured object-oriented code makes it difficult to isolate independent chunks of code for inspection and totally unrealistic to fully comprehend all such chunks in isolation. The following chapters present further investigations that attempt to address the issues facing object-oriented code inspection.

Chapter 4

Systematic, Abstraction Based Object-Oriented Code Inspection

Through the examination of the literature, an empirical experiment, and a survey of industrialists it has been established that some of the key features of object-oriented languages – inheritance, dynamic binding, polymorphism, and small methods – may have a significant impact on the ease of understanding of the resulting program code. These object-oriented features, by distributing closely related information throughout the code, create the problem of delocalisation – the information required to understand one line of code, a method, or even a class is not completely contained within the code under inspection, but spread throughout other methods, classes, systems, or libraries.

Well-structured object-oriented code makes it difficult to isolate independent chunks of code for inspection and totally unrealistic to fully comprehend such chunks in isolation. Based on the results of the previous experiment and industrial survey, three areas were highlighted as needing attention to improve object-oriented code inspections:

- Chunking – how to partition a system for inspection
- Reading strategy – how to read each 'chunk'
- "Localising the delocalisation" – how to make available necessary non-local information

This chapter presents a systematic, abstraction-based reading strategy for object-oriented code inspection that concentrates on addressing the latter two points. Due to the relatively small size of the code inspected and time constraints, the problem of chunking is not explicitly addressed in this experiment. How this problem may be dealt with when scaling-up the approach is discussed at the end of the next section.

A description of the systematic reading strategy is presented along with an empirical evaluation, which takes the form of a controlled experiment comparing the defect detection rates of systematic reading versus the ad-hoc reading strategy.

4.1 Systematic Inspection

The motivation for the proposed technique was the need to address the challenge of "localising the delocalisation" – i.e. to find a way to resolve the references to non-local information by providing many of the benefits associated with thorough systematic reading (accuracy and completeness of information), but in an efficient manner.

The basic approach directs inspectors to read the code in a well-defined order, and as they do this, to reverse engineer abstract specifications for each method. Inspectors follow method calls and other outside information, where necessary, to develop a sufficient understanding. The essential idea is that the creation of abstractions forces a deeper understanding of the code and provides a summary of the method for reference in future inspections. The creation of abstractions is not seen as a duplication of work as it is not common to find class specifications with that level of detail being generated by the design process.

The systematic technique attempts not to place unrealistic constraints on what support documentation is available. It only assumes that code, the Java online API documentation, and class diagrams are available.

The following describes the technique in more detail:

- Interdependencies (couplings) within the whole system are analysed and those classes with least dependencies are inspected first.
- Methods within classes are analysed and those methods with least dependencies are inspected first.

This gives the order in which to inspect the classes in the system and, for each inspection, the reading order for each method. During the inspection:

- Classes and methods are inspected using an abstraction driven reading strategy. This involves reverse engineering an abstract specification for each method.
- During inspection any references to external classes whose understanding is necessary to write the abstract specification must be traced. This may involve reading and understanding other methods, documentation, or previously created abstractions.

As the inspection of the overall system proceeds more and more of the classes will already have abstract specifications. This should limit the need to spend time understanding other classes during future inspections.

The ordering of methods within classes is based on the following features, ranked in order of increasing delocalisation:

- Method call to method previously inspected - including class library.
- Method has a parameter that is a type defined by another class.
- Method casts an object to a type defined by another class.
- Method call in class currently under inspection.
- Method call to class library method not previously looked at.
- Method call outside current class, but in other classes under inspection.
- Method call outside current class and not under inspection.

The ranking was developed by the author through experience gained while creating the systematic reading technique. It may not be possible in all situations to generate a single, unique ordering. Where this occurs, a best fit approach should be taken.

To develop the abstract specification a deep understanding of each method is required. All aspects of the method should be read and understood. All links to other classes should be understood as far as possible. Development of this deeper understanding may reveal more of the hard to find defects.

The systematic technique does not emphasise Soloway's tracing the "flow of the entire program", as this would be impractical given the dynamic characteristics of object-oriented software. It might be impossible in some situations to be able to read all methods when following the trail of delocalisation – there may be too many. In these cases, the trail should be followed until a sufficient understanding has been obtained to allow the abstract specification to be written.

The abstract specification for each method should identify any changes of state (i.e. changes to attributes / instance variables) and outputs (return values or messages) in terms of inputs and prior state (i.e. changes to attributes / instance variables). These are more than just interface descriptions.

The specification generated should be:

- Brief (as short as possible while capturing all aspects of the method).
- Declarative (describe what the method does, not how it does it) and there should be no mention of programming language constructs (e.g. `if` or `while`) and no mention of temporary variables.
- Complete (cover all aspects of the method's functionality including that derived from references to other classes, including inheritance).

The `UserCollection` maintains a list of the people currently registered for the system. People can be added to or removed from the collection. A check can also be performed to see if a person is a registered user of the library system.

Figure 4.1 – Specification for *UserCollection* class

What follows is a brief example (containing one defect) showing the process of writing an abstract specification for the method `isRegistered` from a `UserCollection` class. The specification for the `UserCollection` class is shown in Figure 4.1, with the code for the `isRegistered` method shown in Figure 4.2.

When reading the method, the inspector needs to be aware of the delocalisation that exists within it. These are issues that require further understanding in order to develop the abstraction. In this example, *some* of the delocalisation issues are:

- Uses `Vector` method `elementAt(int)` – what does this do and what type does it return?
- Uses `Person` method `getEmail()` – what does this do and what type does it return?
- Uses method `equals(String)` associated with result of `Person.getEmail()`. Is this defined or is it inherited from `Object`?

```
public boolean isRegistered(String e)
{
  boolean found = false;
  for (int i=0; i< theUsers.size() & !found; i++)
    if ((((Person)theUsers.elementAt(i)).getEmail()).equals(e))
      found = false;
  return found;
}
```

Figure 4.2 – Java code for *isRegistered* method

Inspectors can inspect the code for the method in whatever way they choose – sequentially, inside out, etc., but must resolve delocalisation when encountered. The following shows how an inspector can build up an understanding of the method following a stepwise reading approach. Linger, Mills and Witt [59] developed the stepwise abstraction technique of reading in the late 70’s. Laitenberger [51] has also used a similar approach as part of a code analyst perspective aimed at procedural code.

- `((Person)theUsers.elementAt(i)` gets the ith element from the vector `theUsers` and casts it to a `Person` instance. Can all users be cast to `Person`?
- `(((Person)theUsers.elementAt(i)).getEmail())` gets the email (a `String`) of the ith element in the vector `theUsers`.
- `(((Person)theUsers.elementAt(i)).getEmail()).equals(e))` Compare the input `String e` with the email of the ith element in the vector using `String equals()`, which returns `true` if the two `String` instances consist of identical characters.
- `for` loop iterates through all elements in the vector (0 to size() –1) only while the `boolean found` remains false.
- Loop iterates through the vector an element at a time, while there are elements remaining and the `boolean found` remains `false`, setting the `boolean found` to `false` [sic] if the input `String e` consists of the same characters as the email of the current `Person` object in the vector.

Returns false if the input String e matches the email address of one of the Person elements in the user collection, otherwise returns false.

Figure 4.3 – Final abstraction

From all of this, a final abstract specification can be written for the `isRegistered` method, and is shown in Figure 4.3. This may be a slightly simplistic example, but it highlights how the process of abstraction may encourage the inspector to develop a greater understanding of the code, making it less likely that assumptions or misinterpretations are made. Further examples of abstract specifications can be found in the lecture material used to present the systematic technique (see Appendix C.4).

An important consideration is how the proposed technique would scale up to deal with large amounts of program code. The general guidance in the literature is that limits should be placed in both the amount of code in any one inspection (to around 200 lines of non-commented code) and the time allocated (to around two hours) [31], [36], [96]. Following these guidelines means larger amounts of code must be partitioned or ‘chunked’. This should be carried out with care in order that interdependencies are minimised. The systematic technique proposed would attempt to partition the system into chunks that minimised interdependencies, ideally not splitting a class over more than one inspection. Classes should be ordered so that those with least interdependencies are inspected first. As inspections progress more and more abstractions are generated – ideally saving the inspector the effort of chasing delocalisation (by only reading the abstractions). It is worth noting that no other inspection technique proposes a method that addresses the issue of partitioning large amounts of code into 'inspectable' chunks.

4.2 An empirical Study of Systematic Object-Oriented Inspection

4.2.1 Introduction

In an attempt to evaluate the systematic, abstraction-driven inspection reading strategy, a further controlled experiment was designed that compared its defect detection capability with that of ad-hoc reading.

The following sections present the design of the experiment, the results obtained, and an interpretation of those results. A copy of all the material used for the actual experiment, including details of the defects used can be found in Appendix C.

4.2.2 Experimental Goals and Hypotheses

The aims of the experiment were focused using the Goal Question Metric (GQM) paradigm as described by Solingen and Berghout [87]. The GQM shifts the emphasis away from metrics to goals. The goals create a focus for the important issues of the experiment. These goals are then specified in more detail by defining questions, which in turn suggest the appropriate metrics to be measured. With goals for an experiment stated explicitly, then data collected and the evaluation of that data are based on well-specified rationale. This makes sure that all the necessary information is collected and that all measures required are being made – a lesson learned from the first experiment where not enough information was recorded to allow an accurate description of subject reading strategy.

The style used here to describe the following experimental goals is based on that found in Solingen and Berghout [87].

Goal 1

Analyse the effectiveness of ad-hoc and systematic technique **for the purpose of** comparison **with respect to** their detection of defects **from the viewpoint of** a researcher **in the context of** a University lab course using Java.

This is the main goal of the experiment, evaluating the suggested systematic technique as an aid for defect detection during inspection of object-oriented code. To meet this goal requires answering the following question:

Q1.1: Is there any difference in the number of defects found by either ad-hoc or systematic inspection?

This question may be answered by collecting data for the following metrics:

M1.1.1 Number of defects found, classified by inspection technique

Testable hypotheses are derived from the statement of goals, the questions and the metrics as follows:

H1: The null hypothesis can be described as:

H_0: There is no significant difference in the number of defects found by those subjects performing ad-hoc inspection compared to those performing systematic inspection of object-oriented code.

The alternative hypothesis, H_1, is:

H_1: There is a significant difference in the number of defects found by those subjects performing ad-hoc inspection compared to those performing systematic inspection of object-oriented code.

Goal 2

Analyse the effect of delocalisation **for the purpose of** understanding **with respect to** subjects reading strategy **from the viewpoint of** a researcher **in the context of** a University lab course using Java.

This second goal of the experiment is more exploratory and is aimed at further investigating the nature of delocalised defects and their affect on the reading strategy for the inspection of object-oriented code.

Meeting the above goal requires answering the following questions:

Q1: What way did subjects read through the code?

These questions may be answered by collecting data for the following metrics:

M1.1: Order that classes/methods were read

M1.2: Reading strategy used

Testable hypotheses are derived from the statement of goals, the questions and the metrics as follows:

H2: --: No testable hypothesis – results explored via qualitative analysis.

4.2.3 Experimental Plan

Since the ad-hoc reading technique lacks any explicit methodology, it was chosen as the baseline technique with which to compare defect detection results for the systematic technique.

To investigate the two inspection reading techniques required two groups of subjects to inspect a single code document, one using the ad-hoc reading approach, the other using the systematic approach. This was achieved by assignment of 64 subjects into two groups, A and B, of approximately equal ability (based on previous programming courses). To rule out any interference in the results due to subject ability, the subjects had to inspect a second code document, this time using the alternative approach. Both code documents were similar in terms of size and complexity. Table 4.1 shows the allocation of groups to code documents for the inspection experiment. No group component was carried out, as the main focus of the experiment was the performance of the individual inspectors. The code inspections were paper-based, no tool support was provided (to avoid introducing confounding factors into the experiment).

	Ad-hoc inspection	Stepwise inspection
Group A	Code Document 1	Code Document 2
Group B	Code Document 2	Code Document 1

Table 4.1 – Allocation of groups to code documents

Subjects

Subjects were participants in a 3rd year Honours Computer Science Software Engineering course run at Strathclyde University. It should be noted that these subjects were a completely different set from the first experiment. Subjects had previous experience with the programming languages Java and C, had limited knowledge of Software Requirements Specification (SRS) document inspection, and no experience with code inspections.

Prior to the experiment, subjects had been given a problem statement describing a hotel booking system (the original problem statement can be found in Appendix C.1). From this initial specification, subjects were given six weeks to derive a specification for the system.

Once completed, subjects were then provided with a specification prepared by the course lecturer. From this, subjects were given a further six weeks to code the hotel booking system using Java. It was after this stage in the course that the experiment took place.

Code

Java was used again because the experiment required an object-oriented language and it was the language most subjects knew the best (having used it for the preceding 2.5 years).

In this experiment subjects were required to inspect code segments that were of the order of 200 lines in 90 minutes, bearing in mind that the subjects were students. The amount of code inspected is in line with established practice (see Section 3.2.3 - Code).

For the practice sessions of the experiment, subjects were presented with material used in the first experiment. For the remaining sessions of the experiment, the material used represented extensions onto the hotel booking system. The two extensions were a gym booking facility (code document 1, consisting of one Java class) and a conference room booking facility (code document 2, consisting of three Java classes). The two extensions were of similar length and complexity. Subjects had not previously seen any code documents or specifications for these extensions.

Defects

The defects used were derived from a number of sources: defects with similar characteristics to those used in the first experiment, the literature, the industrial survey, and a selection of naturally occurring defects (defects discovered in the code written by the course lecturer). In total ten defects were seeded into code document 1 and ten different defects into code document 2. Since the experiment was investigating the effects of delocalised defects, half the defects seeded (five defects) in each code document had delocalised features.

Web Material

As well as the paper-based material provided for the inspection, extra material was made available to inspectors via a local web page. This page contained links to the following:

- All code under inspection
- All available code for the rest of the hotel system
- The Java class library API page
- The original hotel system specification

- Abstractions for other system classes that would have already been inspected had the overall strategy of inspecting those classes with least dependencies first been followed (only available for systematic inspections)

All code made available was in plain text and contained no special highlighting, comments or hypertext links.

Data Collection

For ad-hoc inspections, inspectors were provided with a defect report form in which to record defects found and a code booklet containing the code to be inspected. To record subjects' reading order, a collection of boxes were placed above each method in the code documents. Each time a subject began to read a method, they would write the time in the next available box (an example of this can be seen in Figure 4.4).

```
// [   :   ]  [   :   ]  [   :   ]  [   :   ]  [   :   ]  [   :   ]  [   :   ]
  public boolean reserve (Delegate del, int num, FunctionDate start,
                               FunctionDate stop, Set wantedFacilities)
  {
     Function f;
     if (this.isReserved(start, stop) | this.isNameUsed(del.getName()) )
        return false;
     else
     {
        f = new Function(del, num, stop, start, wantedFacilities);
        del.setFunction(f);
        return true;
     }
  }
```

Figure 4.4 – Example of code and time boxes

A questionnaire was prepared and given to subjects upon completion of the ad-hoc and systematic inspections. The aim of the questionnaire was to gather extra information on resources used and problems encountered by subjects during their inspections. A copy of the questionnaires can be found in Appendix C.2.3 (for ad-hoc inspection) and Appendix C.5.5 (for systematic inspection).

For systematic inspections, inspectors were given both a defect report form and code booklet (exactly as for ad-hoc inspection) and were also given method specification sheets. These contained boxes in which subjects were to write their abstract specifications for the inspected code. Systematic subjects were also given a questionnaire which, as well as asking about resources used and problems encountered, explored opinions on the systematic technique and perceived advantages/disadvantages compared to ad-hoc inspection.

Data Analysis

The goals of the experiment feature both a testable hypothesis and an exploratory analysis. Where appropriate, SPSS was used to test the experimental hypothesis (to determine whether there was a significant difference in defects found by ad-hoc inspection compared to systematic inspection of object-oriented code) using an independent sample t-test (this was used because the two groups being compared had different subjects).

The remaining goal that was exploratory in nature was investigated through the analysis of the qualitative information gathered during the experiment and from the post inspection questionnaires.

Threats To Validity

Potential threats to internal validity included:

- Selection effects - subjects were split into two groups of equal ability based on previous class marks in an attempt to minimise this effect as much as possible.
- Learning effects - due to possible learning effects, ad-hoc inspections had to be carried out by both groups of subjects before systematic inspection (necessitating the use of both sets of code each week – see next point).
- Plagiarism was a concern in the experiment since both sets of code documents were used in both the ad-hoc and systematic inspections (weeks 2 and 4), providing an opportunity for collaboration among subjects. This was minimised by retaining all paper material after each experiment. Subjects were also never informed, before or after the experiments, of any specifics about the code being used for the experiment, other than that it was an extension of the hotel booking system.
- Loss of enthusiasm - for four weeks subjects were carrying out an inspection per week. It is possible that subjects found this repetitive and interest dropped off towards the end. To try and counteract this, course credit was awarded to subjects for completing the inspection exercise and the questionnaire.

The potential threats to the external validity were the same as those for the first experiment (use of students, scale of problem inspected, defects seeded and overall inspection process).

4.2.4 Experimental Procedures

Based on the experimental plan, the following timetable was used to arrange the experiment:

Week 1:	Lecture and Practice inspection (using ad-hoc technique)
Week 2:	Inspection of hotel system extensions (using ad-hoc technique)
Week 3:	Lecture and Practice inspection (using systematic technique)
Week 4:	Inspection of hotel system extensions (using systematic technique)

Training for the experiment occurred in weeks 1 and 3, and consisted of an introductory lecture and training session. Each lecture lasted approximately 50 minutes and introduced all of the relevant information and techniques. The next day, a training session lasting 1.5 hours was held and was run informally to allow subjects to ask questions and to overcome any conceptual problems about the inspection process and techniques used.

For the experiment in weeks 2 and 4, subjects were given up to a maximum of 90 minutes to complete the inspection. Subjects were presented with a booklet containing the inspection material (instruction sheet, specification, class diagram, code booklet, defect report form, and method specification sheets for systematic inspections). Once subjects had finished the inspection task or the 90 minutes were up, they were supplied with the questionnaire. Subjects were given approximately 20 minutes to complete this. Both the inspection task and the questionnaire were completed under exam conditions to ensure that subjects worked independently.

4.2.5 Experimental Results

64 subjects participated in the experiment. Due to reasons of attrition from the practice run the results are based on 53 subjects. Three other defects were discovered for the gym code document (code document 1) which were not originally seeded by the author, but were highlighted by subjects during the inspection. The following sections describe the results of the various elements of the inspection exercises.

Defect Detection

Table 4.2 presents a summary of the defect detection results obtained from all parts of the experiment. It shows that for both the Gym and Conference Room extensions there is a small improvement in the mean number of defects found by subjects using the systematic inspection technique when compared to ad-hoc inspection. Using an independent sample t-test, the difference between the two means (ad-hoc and systematic) is not statistically significant (at the 5% level) for both code documents (shown in Table 4.3). This means we cannot reject the null hypotheses, H_0, for goal 1 (see Chapter 4.2.2): there is no significant difference in the number of defects found by those subjects performing ad-hoc inspection compared to those performing systematic inspection of object-oriented code.

	Gym		Conference Room	
Code Document	1		2	
Technique	Ad-hoc	Systematic	Ad-hoc	Systematic
Group	A	B	B	A
No. of classes	1	1	3	3
No. of Subjects	25	28	28	25
No. of defects in code	13	13	10	10
No. of defects found (mean)	3.44	3.86	3.04	3.44
No. of defects found (St. dev)	2.0632	2.4603	1.7947	1.4166
No. of defects found (St. error)	0.4126	0.4649	0.3392	0.2833
Time (mean) (minutes)	84	88	89	88
False positives (mean)	5.08	3.61	4.71	4.28
False positives (St. dev)	3.4147	2.6852	2.9796	2.0314
False positives (St. error)	0.6829	0.5075	0.5631	0.4063

Table 4.2 – Inspection Summary of defect detection results

Table 4.2 also shows the mean time taken by each group of subjects for their inspection. In general, most subjects, no matter the technique or code document, used the full time of the inspection (90 minutes).

False positives are defects noted during the experiment which turn out not to be defects. The results indicate that for both code documents, there was a reduction in the mean number of false positives recorded by the systematic subjects compared to the ad-hoc subjects. This is also reflected in the standard deviation and standard error results. This may be due to the systematic technique encouraging inspectors to obtain a greater level of understanding. The reduction in false positive figures may also be due to the subjects gaining more experience in inspection.

	Significance (2-tailed)
Gym	.505
Conference Room	.365

Table 4.3 – Results of an independent sample t-test

Shown in Figure 4.5 and Figure 4.6 are the mean defect detection rates for the two hotel system extensions. Each figure compares the average number of defects found by all the ad-hoc inspectors over the 90 minutes with the average number of defects found by all the systematic inspectors.

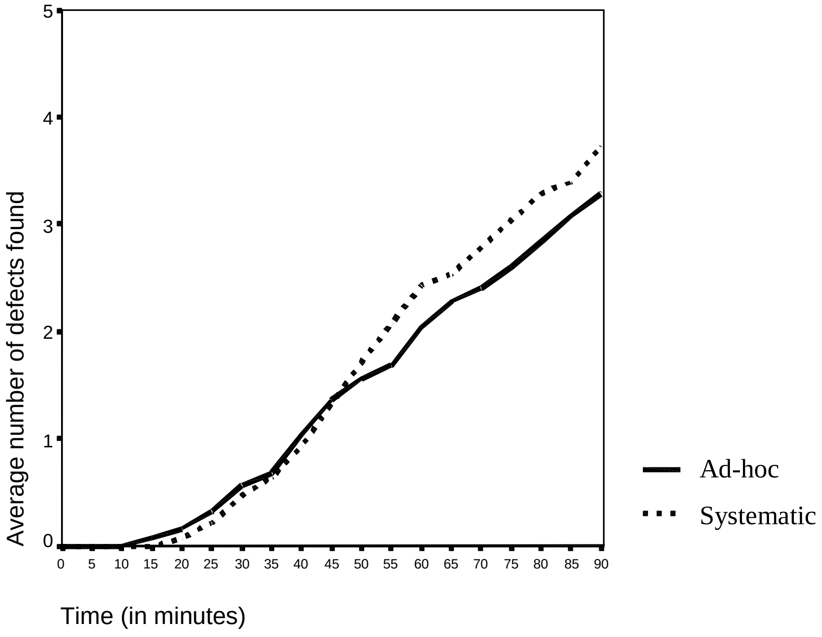


Figure 4.5 – Gym defect detection rate

In Figure 4.6 there is a large difference between the initial defect detection rate. This could be due to the fact that the conference room extension was written in three classes. Following the systematic technique meant reading through the classes in a certain order. The first class they would have read only had one defect and the second class only had three (out of a possible ten). Those who were inspecting the three classes via the ad-hoc method were given the classes in one of six different orderings. This could account for the higher difference between the two techniques when compared to the gym extension in Figure 4.5 (which only had one class and so no alternative orderings), where the defect rates for the two techniques are fairly close. For both code documents, the systematic technique at some point obtains a better detection rate than the ad-hoc technique.

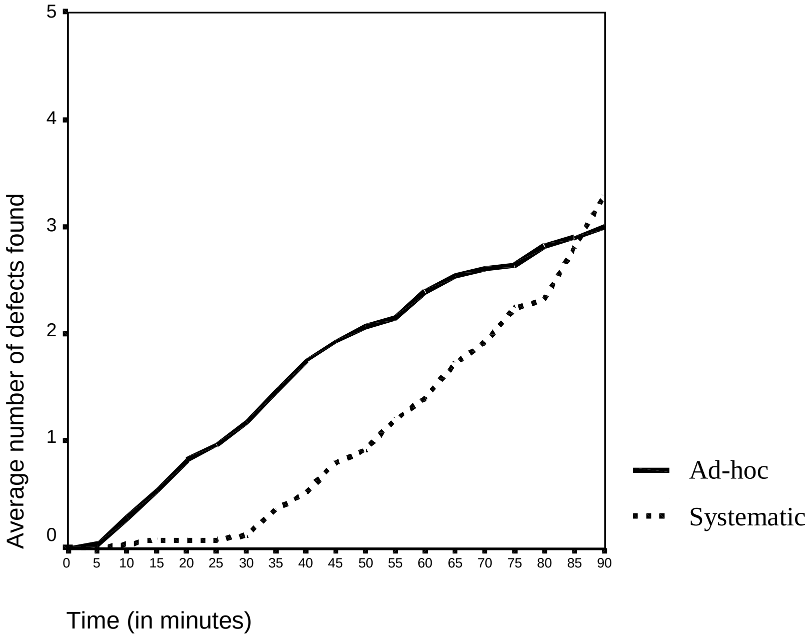
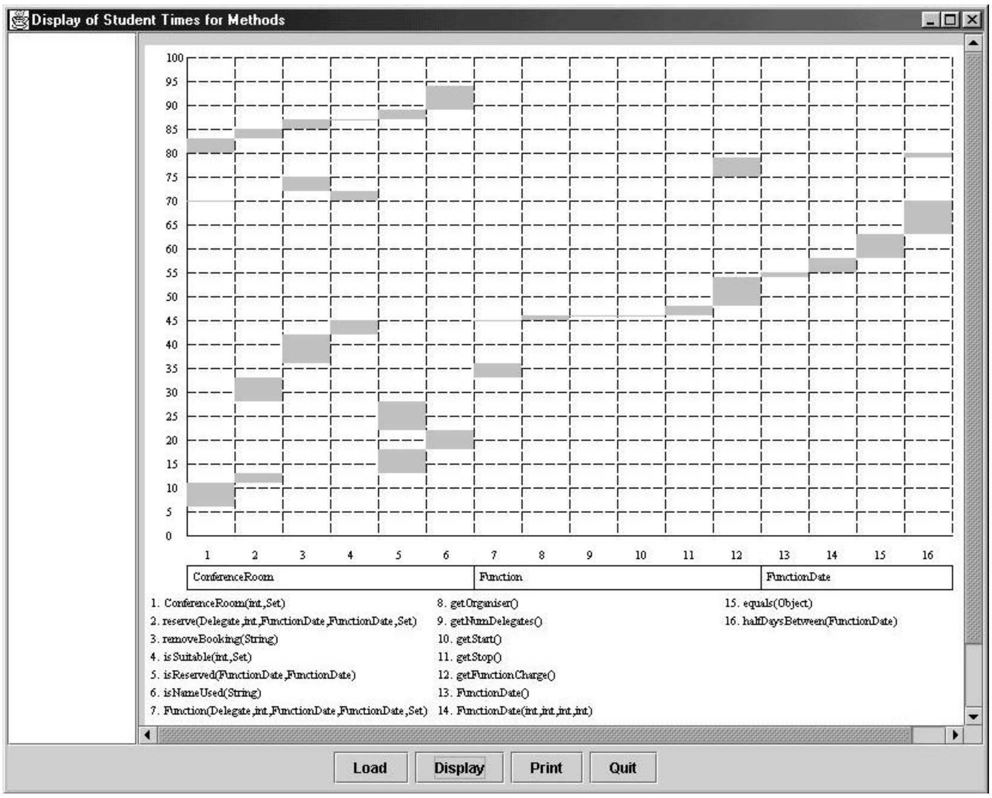


Figure 4.6 – Conference Room defect detection rate

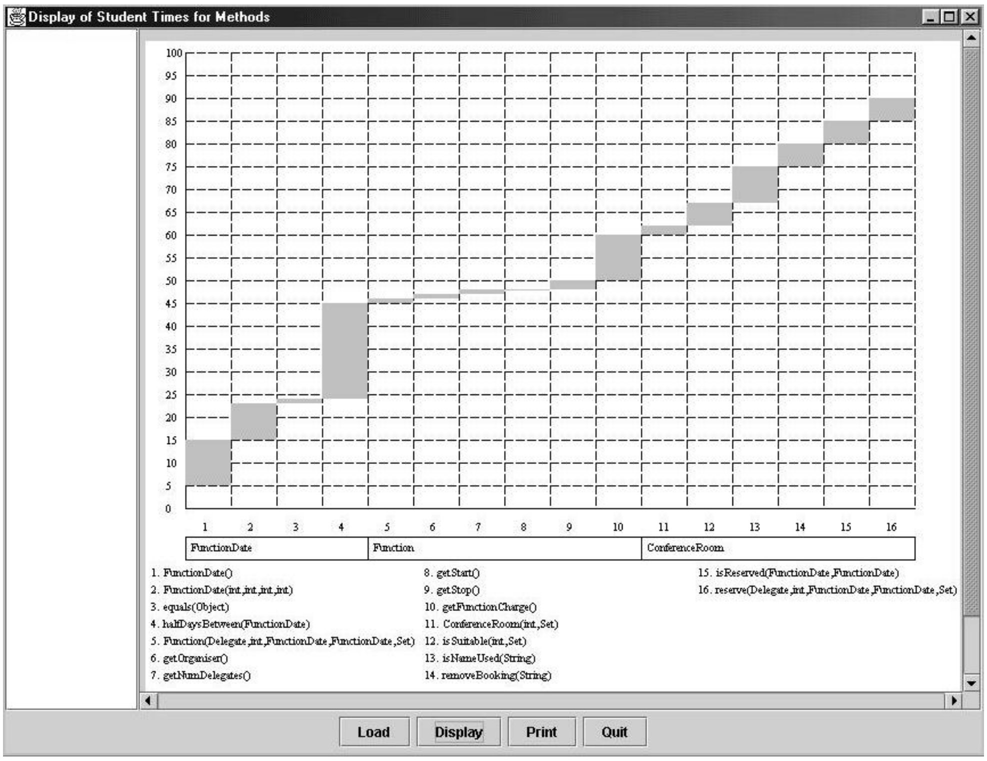
Reading Strategy

This section investigates the second goal of the experiment, looking at the reading strategy employed by subjects when reading the code. To help gauge subjects' reading strategy, they had to fill in the time they began reading a method in boxes provided within the code documents (an example is shown in Figure 4.4). This information was then compiled and entered into a small purpose built tool to help visualise how subjects read through the code. A screen captured from the tool showing an ad-hoc subject's reading order is shown in Figure 4.7. It shows in what order the classes and methods were read as well as approximately how long was spent reading them. This particular subject read through the code using a combination of two techniques. The subject began by reading the code in the order presented to them, but then would follow method calls to other methods within the class or methods in other classes. Figure 4.8 shows an example of a systematic subjects' reading order. This subject read the code in the order presented to them, and read each method only once.



1. ConferenceRoom(int,Set)
2. reserve(Delegate,int,FunctionDate,FunctionDate,Set)
3. removeBooking(String)
4. isSuitable(int,Set)
5. isReserved(FunctionDate,FunctionDate)
6. isNameUsed(String)
7. Function(Delegate,int,FunctionDate,FunctionDate,Set)
8. getOrganiser()
9. getNumDelegates()
10. getStart()
11. getStop()
12. getFunctionCharge()
13. FunctionDate()
14. FunctionDate(int,int,int,int)
15. equals(Object)
16. halfDaysBetween(FunctionDate)

Figure 4.7 – Screen shot of an ad-hoc subject's reading strategy



1. FunctionDate()
2. FunctionDate(int,int,int,int)
3. equals(Object)
4. halfDaysBetween(FunctionDate)
5. Function(Delegate,int,FunctionDate,FunctionDate,Set)
6. getOrganiser()
7. getNumDelegates()
8. getStart()
9. getStop()
10. getFunctionCharge()
11. ConferenceRoom(int,Set)
12. isSuitable(int,Set)
13. isNameUsed(String)
14. removeBooking(String)
15. isReserved(FunctionDate,FunctionDate)
16. reserve(Delegate,int,FunctionDate,FunctionDate,Set)

Figure 4.8 – Screen shot of a systematic subjects' reading strategy

Figure 4.9 shows a graph detailing number of times the code to be inspected was read against percentage of subjects. It shows that those carrying out ad-hoc inspections were always reading the code more than once, and in nearly 50% of the cases read the code two to three times. In comparison, 30-40% of subjects using the systematic reading technique read the code only once, with the vast majority of the rest reading through the code once and then re-reading only a few select methods. When inspecting via the systematic technique, subjects were spending longer reading methods as they attempted to fully understand what they were doing and create their abstract specifications. Whereas with ad-hoc inspections, subjects would repeatedly browse through the code multiple times, sometimes appearing to spend very little time concentrating on each method.

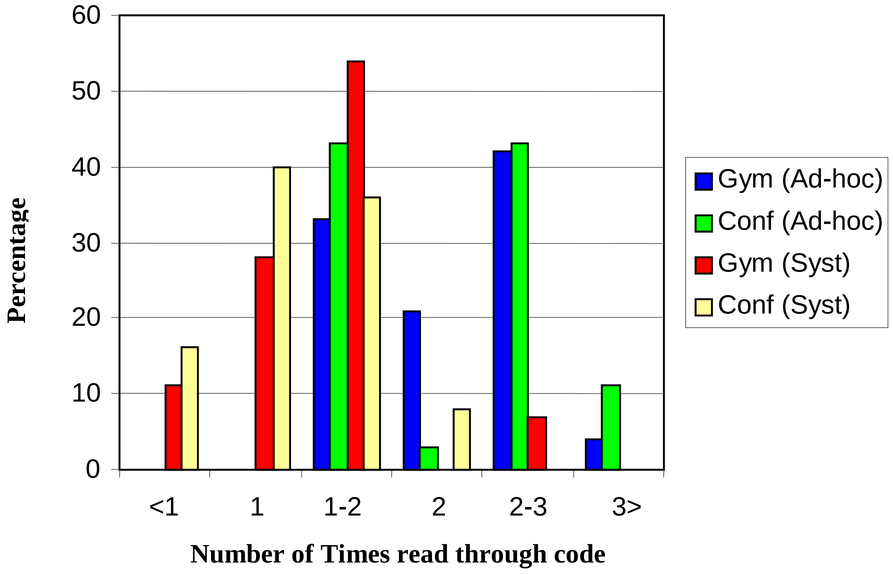


Figure 4.9 – Number of times subjects read through the code

Further analysis of the timing information gathered indicates that more than half of the subjects in both ad-hoc inspection groups began reading the code in the order presented to them. After reading through the code at least once, subjects then used this information to revisit methods for further inspection. It appears that in the later stages of the inspection, the subjects' reading order is not affected by code order or method calls.

Most of the remaining ad-hoc subjects read the code by following method calls. After having read through all the code this way, at least once, subjects again decided which methods to revisit.

Only 17% of all ad-hoc subjects read through the code in the order presented and stuck to that reading order through the entire inspection.

For the systematic inspection, subjects were told to read the code in the order presented to them (for both methods and classes). The code had been specifically ordered to minimise dependencies. About half read the code in the suggested order for the entire inspection, the other half read most of the code in the order suggested, but occasionally jumped to another method before returning to the given order. More time was spent by all subjects reading methods for the first time than was the case with ad-hoc inspection.

Through looking at the timing information gathered and the defects found by subjects, it appears that 9% of the systematic subjects failed to complete their inspection of the code (i.e. were not able to read all the methods in the code within the given time).

	More than half the defects found on first run through code	More than half the defects found on subsequent runs through code	Same number of defects found on first and subsequent runs
Gym (ad-hoc)	54%	21%	25%
Gym (syst)	75%	4%	21%
Conference (ad-hoc)	54%	32%	14%
Conference (syst)	100%	0%	0%

Table 4.4 – When, during the inspection, subjects found defects (in relation to their reading strategy)

Table 4.4 shows when subjects were more likely to find defects during the inspection. For those carrying out ad-hoc inspection, just over 50% were finding more than half of their total defects in their first pass through the code. Roughly a quarter of the remaining subjects found more defects on subsequent runs through the code. This is not surprising considering that ad-hoc inspectors were making multiple passes through the code (see Figure 4.9). For systematic inspection, 75% of subjects for the single class code document (gym) and 100% of subjects for the multiple class code document (conference) found more than half their defects on the first pass through the code. Very few subjects in either group found more defects on subsequent passes through the code. This significant increase in finding more defects in the first pass through the code indicates that there is perhaps not quite such a strong need to make the multiple passes through the code (as seems to be the case with ad-hoc inspection – shown in Figure 4.9), but fewer, more concentrated and focused passes.

While carrying out their inspections subjects had access to the online Java documentation. The post-inspection questionnaire asked subjects what online documentation they had accessed. Results show that very few subjects (in most some cases no more than 25%) who were reading the relevant online documentation associated with a defect were also finding the defect. The online documentation was the standard JavaDoc documentation supplied with Java. Subjects had access to it for the previous two years. The results suggest that either subjects are not very proficient at using the online documentation, or the online documentation itself is at fault (perhaps due to a lack of detailed information or poor presentation). No definite conclusions can be made.

Code Document (ad-hoc inspection)	No. of defects found (ad-hoc)	Code Document (systematic inspection)	No. of defects found (systematic)	% of defects (comparing both inspections)
Gym	7 / 13	Conference	6 / 10	54% : 60%
	7 / 13		2 / 10	54% : 20%
	6 / 13		5 / 10	46% : 50%
	6 / 13		4 / 10	46% : 40%
	6 / 13		4 / 10	46% : 40%
	6 / 13		5 / 10	46% : 50%
Conference	7 / 10	Gym	9 / 13	70% : 69%
	6 / 10		6 / 13	60% : 46%
	6 / 10		7 / 13	60% : 54%
	6 / 10		8 / 13	60% : 62%
	5 / 10		4 / 13	50% : 31%
	5 / 10		5 / 13	50% : 38%

Table 4.5 – Number of defects detected by the top ad-hoc subjects and their systematic defect detection results

Table 4.5 shows for each of the ad-hoc subjects who performed well in either the gym or conference code documents, the number of defects the subjects went on to find using the systematic technique. Those subjects who performed well during the ad-hoc inspection did not significantly improve their performance when carrying out the systematic inspection. Eight of the twelve subjects obtained approximately the same percentage of defects for both inspections, the other four subjects performed notably worse during the systematic inspection than in the ad-hoc inspection. This possibly indicates that the systematic technique was constraining the natural abilities of the better subjects, perhaps by forcing them to read the code in a certain order. It is also possible that these subjects felt they had less freedom to look back at code already inspected, instead always reading forward through the code in order. Conversely, Table 4.6 shows for nine of the poorest subjects in ad-hoc inspection, all but one improved their defect detection rate during the systematic inspection. The application of a technique to guide the inspection process appears to help those weaker subjects.

Code Document (ad-hoc inspection)	No. of defects found (ad-hoc)	Code Document (systematic inspection)	No. of defects found (systematic)	% of defects (for both inspections)
Gym	0 / 13	Conference	2 / 10	0% : 20%
	1 / 13		4 / 10	8% : 40%
	1 / 13		1 / 10	8% : 10%
	1 / 13		3 / 10	8% : 30%
Conference	0 / 10	Gym	2 / 13	0% : 15%
	0 / 10		2 / 13	0% : 15%
	1 / 10		4 / 13	10% : 31%
	1 / 10		0 / 13	10% : 0%
	1 / 10		2 / 13	10% : 15%

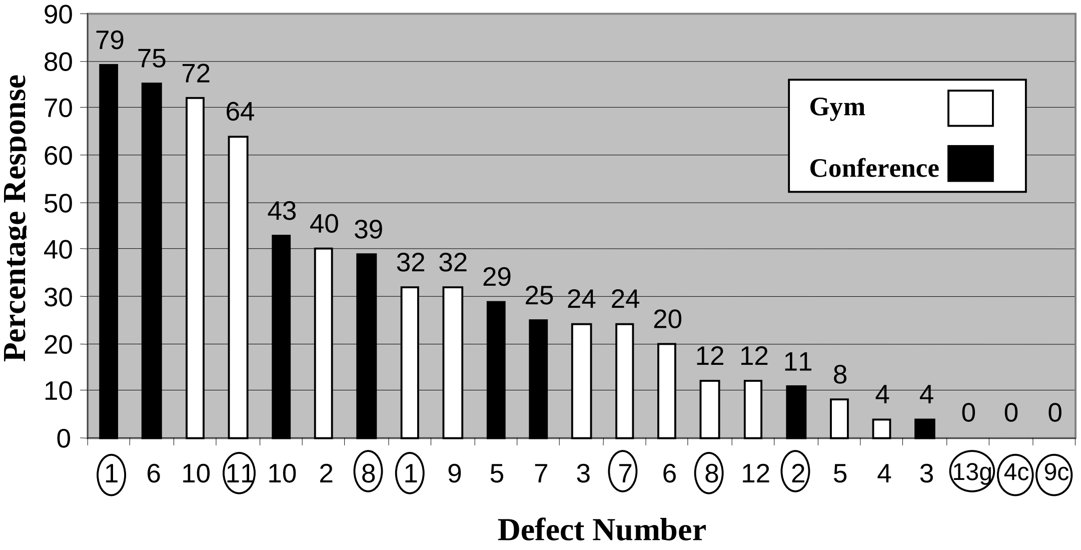
Table 4.6 – Number of defects detected by the worst ad-hoc subjects and their systematic defect detection results

Just under half of those subjects who performed well (shown in Table 4.5) during ad-hoc inspection (over both code documents) read the code by following method calls, the rest read the code in the order it was presented to them. All of the subjects who did not perform as well during the ad-hoc inspection (shown in Table 4.6) read the code in the presentation order.

The Defects

The two charts in Figure 4.10 show the percentage of subjects (y-axis) who found each particular defect (x-axis) and which code document the defect belonged to (colour of the bar) for each of the two inspection techniques. This clearly shows which defects were discovered relatively easily and those that were harder to identify. For the ad-hoc inspections, the three defects that were not found by any subjects all have delocalised features (all were found by systematic inspectors – Defect 4c by 32%, Defect 9c by 12%, and Defect 13g by 18%). Most of the remaining delocalised defects were found by less than 39% of subjects. Defects 4, 5 and 13 for the gym code document were not seeded by the authors, but were found by subjects during the inspection. One of those, defect 13, was only found by the systematic inspection technique.

Ad-Hoc Defect Results



Note: For last three defects, code base highlighted by letter, e.g. g - gym, c - conference

○ - delocalised defects

Systematic Defect Results

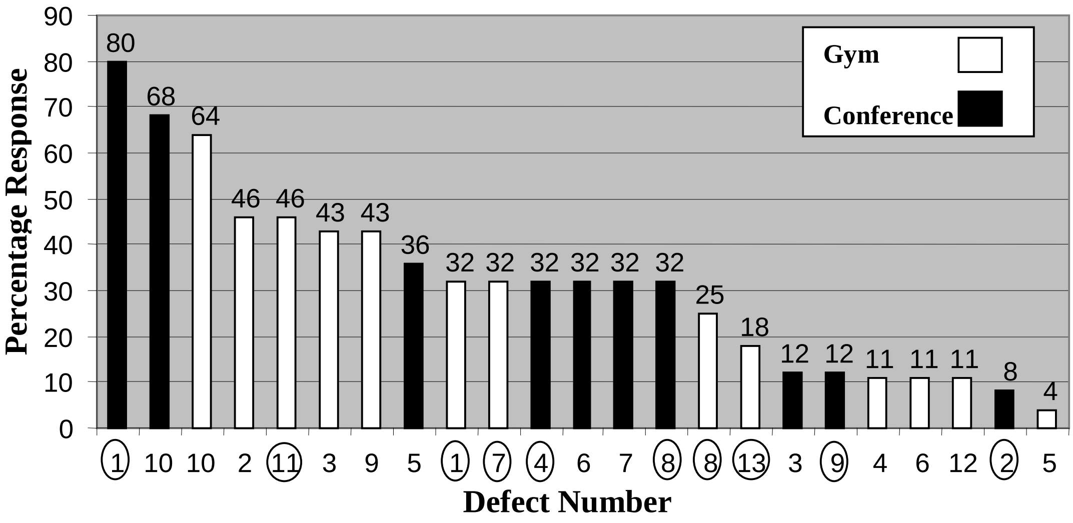


Figure 4.10 – Percentage of subjects finding each defect for both code documents and defect detection techniques

Defect No.	1	6	10	11	10	2	8	1	9	5	7	3	7	6	8	12	2	5	4	3	13	4	9
Locality (M,C,S)	S	S	M	S	M	C	S	S	C	C	M	C	S	M	S	C	S	M	C	M	S	S	S
Method (S,M,L)	L	M	L	L	M	L	M	L	M	M	M	L	L	L	L	L	M	L	L	M	L	M	L
Alg/Comp			X		X	X			X	X	X	X		X		X		X	X	X	X		X
Use of class library				X			X	X					X		X		X					X	X
Wrong object							X							X	X					X			
Wrong message	X	X					X	X					X		X		X					X	
Data flow error				X							X			X		X		X	X	X			X
Instance variable														X			X			X			
Specification clash	X		X			X				X	X	X											
Omission						X				X	X	X		X				X		X			X
Commission	X	X	X	X	X		X	X	X				X		X	X	X		X		X	X	
Delocalised	X			X			X	X					X		X		X				X	X	X
%	79	75	72	64	43	40	39	32	32	29	25	24	24	20	12	12	11	8	4	4	0	0	0

Gym	□
Conference	▨

Figure 4.11 - Defect characteristics in percentage response order (both code sets) – Ad-hoc inspection

Gym □

Conference ▨

Defect No.	1 ▨	10 ▨	10	11	2	9	3	5 ▨	8 ▨	7 ▨	6 ▨	4 ▨	7	1	8	13	9 ▨	3 ▨	12	6	4	2 ▨	5
Locality (M,C,S)	S	M	M	S	C	C	C	C	S	M	S	S	S	S	S	S	S	M	C	M	C	S	M
Method (S,M,L)	L	M	L	L	L	M	L	M	M	M	M	M	L	L	L	L	L	M	L	L	L	M	L
Alg/Comp		X	X		X	X	X	X		X						X	X	X	X	X	X		X
Use of class				X					X			X	X	X	X		X					X	
Wrong object									X						X			X		X			
Wrong message	X								X		X	X	X	X	X							X	
Data flow error				X						X							X	X	X	X	X		X
Instance variable																		X		X		X	
Specification	X		X		X		X	X		X													
Omission					X		X	X		X							X	X		X			X
Commission	X	X	X	X		X			X		X	X	X	X	X	X			X		X	X	
Delocalised	X			X					X			X	X	X	X	X	X					X	
%	80	68	64	46	46	43	43	36	32	32	32	32	32	32	25	18	12	12	11	11	11	8	4

Figure 4.12 - Defect characteristics in percentage response order (both code sets) – Systematic inspection

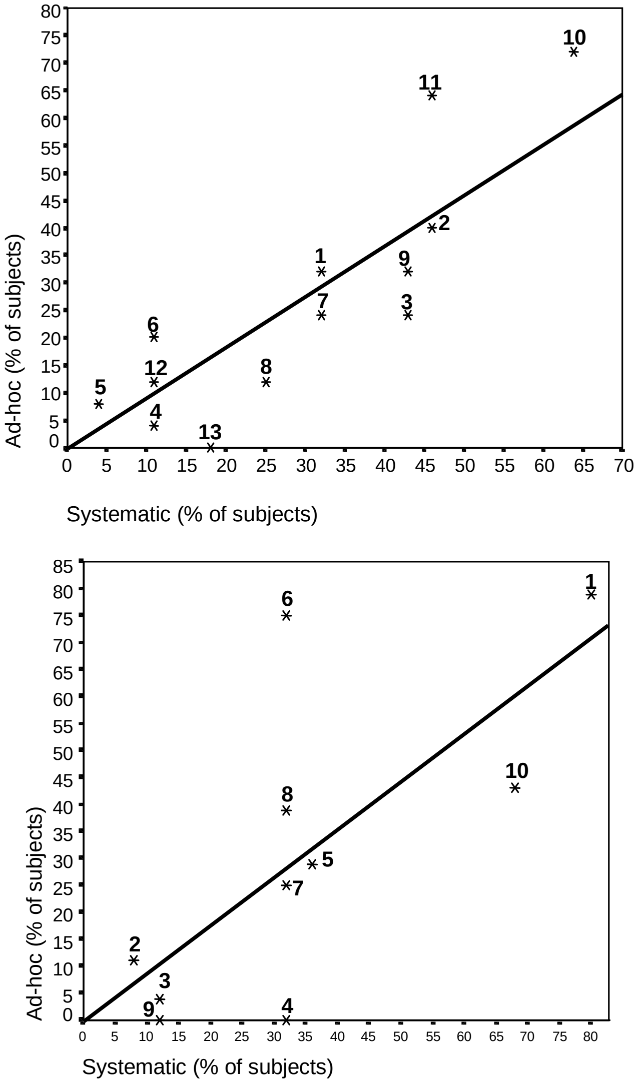
One delocalised defect, defect 1, in the ad-hoc inspection of the conference code document was found by 79% of subjects (see top picture in Figure 4.10). The defect concerned a method in the `Function` class calling an incorrect method (`daysBetween` in the `HotelDate` class instead of `halfDaysBetween` in the `FunctionDate` class). This high response rate may have been due to a clash with the class specification or class diagram provided to subjects during the experiment, or it may be that the subjects noticed there was a method unused in one of the related classes they were inspecting (`halfDaysBetween` in the `FunctionDate` class).

For each of the defects (23 in total) over both code documents, a list was drawn up of their characteristics, similar to those used in the previous experiment (see Figure 3.7). Figure 4.11 and Figure 4.12 show for both reading techniques, the characteristics for the defects in percentage response order. The characteristics listed were similar to those used in the previous experiment but included two other headings: specification clash (defect highlighted by clash with specification) and delocalised (defect contains characteristics which made it delocalised in nature).

The defect characteristic information from Figure 4.11 and Figure 4.12 was again entered into C5.0, a rule induction system (the output generated can be found in Appendix C.7). The following points were observed from the rules that were generated:

- For both inspection methods, defects involving wrong object and instance variable misuse were difficult to find. Also, defects involving data flow errors and class library access were for the most part difficult to find, with one or two exceptions.
- There were slight improvements from ad-hoc to systematic inspection for defects with wrong message and use of class library characteristics.
- Defects involving a clash with the specification were in the higher response end of the tables for both inspection methods.
- Defects involving omission were never found by more than 46% of the subjects and proved difficult for both inspection techniques.
- There is a slight rise in response rates via the use of the systematic technique. The different gradients for the two techniques can be seen in Figure 4.10. For the top graph, the gradient is fairly steep, hitting percentages in the 20’s by mid way and ending on a zero response rate. The bottom graph, representing the systematic results shows a curve (after a sharp initial drop) with a more gradual decline, by half way still having percentages in the 30’s, and not ending on a zero response rate.

Figure 4.13 highlights the frequency of detection of defects in both code documents. Each axis represents the percentage of subjects who found the defects (data points) for a certain inspection technique. Defects of interest are those not near the line, but are nearer the axis lines, indicating that one technique was more successful at finding that defect. There are several defects for both code documents that fit into this category, as the ratio of comparison between the two techniques is 2:1 or worse (defects 3, 4, 5, 6, 8, and 13 for gym, and defects 3, 4, 6, and 9 for conference).



Gym code document

Conference code document

Figure 4.13 – Frequency of detection for defects in both code documents

Of the six defects mentioned for the gym code, defects 8 and 13 were delocalised, appeared in large methods, had system locality (information required to identify the defect is distributed across multiple classes), and both were found more by systematic inspectors. The systematic technique may have helped with these defects due to the increased understanding subjects were encouraged to obtain through the creation of the abstractions, and being forced to follow the trails of delocalisation. From the remaining four, three were related to defects of omission, and one relating to extraneous code. Two of the omission defects were found more by ad-hoc inspectors, the other two defects were found more by systematic inspectors.

Three of the four defects for the conference code were delocalised defects (two dealt with omission, the other using a wrong method from the class library), and all were found by substantially more systematic inspectors than ad-hoc. The remaining defect, defect 6, involved parameters being sent via a method call in the wrong order. This defect was found by substantially more ad-hoc inspectors (more information concerning conference room defect 6 can be found in Chapter 4.2.7, Interpretation of Results).

Questionnaire Results

A questionnaire was given to each subject after the completion of each inspection exercise (for both ad-hoc and systematic reading). A copy of the questionnaires can be found in Appendix C.2.3 (for the ad-hoc inspection) and Appendix C.5.5 (for the systematic inspection). The following summarises the questionnaire responses for each of the inspection techniques.

Ad-hoc inspection

With the ad-hoc inspection technique nearly half of all subjects read through the code in sequential order presented and the other half read through code trying to follow execution path/method calls. Subjects using the ad-hoc inspection technique stated that it was less time consuming (twenty-one subjects), less restrictive in its reading order, e.g. left to their own devices (fifteen subjects) and easier compared to the systematic technique (six subjects). The problems with ad-hoc inspection were that it did not help with understanding (six subjects), presented less structure to the inspector and subsequently there was more jumping around the code (twenty-two subjects). Other problems for one or two subjects that arose during the ad-hoc inspection included dealing with classes from the Java class library that subjects had not previously seen and coding style.

When asked what could be done to improve ad-hoc inspections, subjects' responses included access to tools and aids (two subjects), more practice (four subjects), ordering of code (six subjects) and more structure to the technique (two subjects).

Systematic inspection

With the systematic technique subjects found that reading the code in the suggested order meant there was less jumping around (twenty-two subjects), they gained an improved understanding (fourteen subjects), and that the process was more structured/focused (seven subjects). The down side found by following the ordering was that it took longer to read through the code (five subjects). One or two subjects commented that it could sometimes feel restrictive and major methods were being left to the end of the ordering.

When creating the abstractions, subjects suggested that it encouraged understanding and made you read each line of code (thirty subjects), that instead of having to re-read methods you could read your previously written abstractions (six subjects), and that subtle defects were easier to identify (three subjects). The problems with the abstractions were that they were time consuming (twenty-nine subjects), there was too much to write (five subjects), and that in several cases it was difficult to write natural language specifications based on the code (twelve subjects).

When asked what could be done to improve systematic inspections, subjects' main concerns were more time (six subjects), more practice (three subjects), and more information/examples at the lecture (three subjects).

4.2.6 Experimental Design Lessons

One matter that was very apparent from the questionnaires given to subjects after each experiment was that they desired more practice with inspection. In this experiment subjects were only given one week of practice with a technique before the assessed part of the experiment the following week. Subjects only had access to two examples of the application of the systematic inspection reading technique, one presented in the lecture and one from the practice session. To improve the validity of the experiment it is important that subjects understand and feel comfortable with any reading technique they have to apply. In future experiments there should be more examples and/or more practice sessions for subjects.

4.2.7 Interpretation of Results

The main interpretation of these results is that there is no significant difference between the systematic technique and the ad-hoc technique in terms of the average number of defects discovered (see Figure 4.5 and Figure 4.6), although there is a small improvement for both code samples using the systematic approach. This means we cannot reject the null hypothesis, H_0, for goal 1: there is no significant difference in the number of defects found by those subjects performing ad-hoc inspection compared to those performing systematic inspection of object-oriented code.

On the other hand the results suggest that the systematic technique is no worse than ad-hoc in terms of defect detection and there may be a number of potential benefits from the use of the systematic approach:

a) The systematic approach found all the defects, the ad-hoc approach did not. Ad-hoc inspectors did not find three of the ten delocalised defects (one in gym and two in conference). Although no group component (collation of defects by individual inspectors) was carried out, the fact that the systematic technique found all the defects might suggest that the group component would be more successful.

b) The systematic approach produced abstractions for every method as a by-product. It is intended that these abstract specifications can be used in future inspections to save the inspector, or other inspectors, the effort of reading the class or method again when another class makes a delocalised reference to that class (e.g. via inheritance, variable declaration, method invocation,…). Further research is necessary to investigate the usefulness of these abstract specifications. In particular, it is important to investigate the level of formality required – would the precision and conciseness of semi-formal or formal specifications provide overall benefits in terms of removing ambiguity compared to natural language?

c) There is anecdotal evidence from the subjects' questionnaires that the task of creating abstract specifications encouraged a greater understanding of the code under inspection. It is reasonable to assume that a greater understanding may lead to better defect detection, especially of more subtle defects. The fact that the systematic inspectors found all the defects also provides some support for this view.

d) The systematic approach provides an ordering for the reading strategy to deal with the delocalised, distributed nature of object-oriented software. Again the questionnaire data suggested that inspectors appreciated the rigour imposed by this ordering. Without such an ordering it is possible that inspectors may 'wander off' into the rest of the system chasing a thorough understanding but, without great care, there is a danger that a thorough and complete coverage of the classes under inspection will not be achieved.

e) Related to points c) and d) is the suggestion that the technique helped the weaker subjects improve their defect detection. An analysis of the nine poorest subjects in ad-hoc inspection over the two sets of code documents showed that all but one improved their defect detection rate during the systematic inspection. Alternatively this could be as a result of a learning effect. On the other hand there is similar evidence that the systematic method may have inhibited the natural abilities of the stronger subjects.

Interpreting the results also leads to suggestions for potential refinements to the systematic method:

a) The systematic inspectors tended to make one, or at most two, relatively slow passes through the code. The systematic approach seemed to take time to build up momentum (see Figure 4.6) when inspecting the multi-class code (the Conference room extension). The questionnaire data suggests that subjects found that there was too much to write during systematic inspection and that they found it difficult to write the required natural language specifications. This suggests that there is a need to make the abstracting process more efficient – the abstractions should be as focussed and brief as possible, but balanced against the need for future inspectors to be able to use them as an efficient alternative to reading the class.

b) There appeared to be a real requirement for more training in the systematic approach. Subjects were given a 1-hour lecture and a 2-hour practical session on its application. The questionnaires suggested a need for more lecture examples and more practical experience with the technique. Increased experience with the systematic approach, particularly with the process of creating specifications, may improve the efficiency and effectiveness of the approach.

c) Related to a) is the possibility that tool support may help make the creation of abstract specifications more efficient e.g. by automatically identifying state change variables and output values for which the inspector must write specifications. Several subjects’ questionnaires also suggested difficulties with the variety of documents to be managed during an object-oriented inspection e.g. code sheets, problem specification, class diagram, defect report form, abstraction sheet, as well as having to access the Java Class library API on-line. Again, it is possible that the process may be made more efficient by appropriate tool support.

d) The systematic approach imposed a reading order that minimised interdependencies – basically methods and classes are ordered so that they are read in order of increasing coupling. However the graph-based nature of object-oriented interactions means that all dependencies cannot be read and understood before they are used. The method needs to prescribe how to deal with such situations. For example, one particular defect (defect 6 – Conference code – see Figure 4.10) highlighted this type of problem. It involved the incorrect ordering of parameters in a call to a method. The method being called had already been inspected. If subjects had looked at the method in the other class, or looked at their abstraction sheet, they should have noticed the defect. 75% of ad-hoc inspectors found defect 6, compared to 32% for systematic. The ad-hoc inspectors had more freedom to move around the code. It is possible that the systematic method may have discouraged inspectors from looking back.

A key finding was that ad-hoc inspectors seemed to perform multiple (two or three) passes through the code following a combination of code ordering and tracing dynamic method invocations. This was in contrast to the more methodical, single pass (or so) of the systematic inspectors. In this study the former approach appears to have been as effective at defect detection as the systematic approach. The complete systems were relatively small (a few thousand lines of code). An interesting question for further study is how well the two strategies would cope with a more realistic scenario where inspectors are reviewing 200 line ‘chunks’ from significantly sized object-oriented systems where delocalised references could lead deep into the rest of the system.

One potential weakness of the systematic strategy adopted for this study may be that it is based on a static view of the code. Specifically, the subjects are encouraged to read the code in a linear order (where that order is such that, as far as possible, dependencies are read before they are used). However the dynamic view of object-oriented code is quite different from the static view, as found by Gamma *et al*. [35] (see Chapter 2.2), who stated that the two are largely independent of each other. This makes it very difficult to understand one from the other.

These findings suggest that the systematic approach offers a number of benefits: a rigorous reading strategy, potential to help address delocalisation through abstract specifications, potential to encourage deeper understanding and to discover different defects from an ad-hoc approach. On the other hand the systematic approach doesn’t adequately address the highly dynamic nature of object-oriented software, may be more time consuming and may restrict the natural ability of experienced or skilful inspectors.

A final interpretation of the results is that they provide further confirmatory evidence of the problems caused by delocalisation during object-oriented inspection. Figure 4.10 shows that the delocalised defects are, in the main, to the right (found by less than 39% of subjects). The inductive analysis suggested that characteristics of delocalisation – using the wrong object and class library access – were amongst the characteristics of difficult to discover defects. The results also show that very few subjects who actually read the relevant online documentation actually found the associated defect (in the main less than 25%).

4.3 Conclusions

This chapter described the evaluation of a systematic, abstraction-driven inspection technique for object-oriented code that was developed to address the problems of reading strategy and ‘localising the delocalisation’. No significant difference was found in terms of the number of defects discovered when compared to an ad-hoc method of inspection. However some potential benefits were discovered which, with further refinement of the approach, may help address the problems of delocalisation and provide a suitable reading strategy for object-oriented code. This experiment also uncovered further evidence that the delocalised nature of object-oriented code is a real problem during software inspection.

More research is required to investigate whether refinement of this systematic approach can provide a pragmatic reading strategy that helps address delocalisation (as well as addressing the problem of how to break a large object-oriented system into ‘chunks’ for inspection). On the other hand, it may be that the dynamic nature of object-oriented systems hinders the effectiveness of such a systematic approach. The next chapter describes an investigation of an alternative code reading strategy that is more in tune with the dynamic nature of object-oriented systems, and an investigation of whether refinements of the systematic approach can provide a pragmatic reading strategy that helps address delocalisation.

Chapter 5

Development and Evaluation of Three Techniques for Object-Oriented Code Inspection

Through a controlled experiment three significant issues important to the successful inspection of object-oriented code were identified: chunking, reading strategy, and ‘localising the delocalisation’. From this, a second controlled experiment investigated a systematic abstraction-driven inspection technique developed to address the problems of reading strategy and delocalisation. It was found that the systematic approach offered a number of benefits: a rigorous reading strategy which encouraged a deeper understanding of the code combined with the potential to address delocalisation through the creation of abstract specifications. However, the systematic approach did not appear to address adequately the problem of defects associated with the highly dynamic nature of object-oriented software. The main findings from the second experiments were that delocalisation of information and the difference between the static and the dynamic views appear to be very real problems for the practical application of software inspection to object-oriented code.

This chapter presents three diverse reading techniques that were developed for object-oriented code inspection: an updated version of the systematic technique, a checklist modified to encourage inspectors to develop an understanding of the code and focus more on object-oriented issues, and a use-case driven approach which takes a slice through the system in order to gain a more dynamic view of the code. These techniques are then evaluated through a controlled experiment.

5.1 Three Inspection Reading Techniques

To further address the problems of delocalisation and reading strategy, three reading techniques were developed: a systematic abstraction-based technique; a modified checklist; and a strategy based on use-cases.

The systematic abstraction-driven approach aims to support an understanding of the code in a rigorous, but sequential, fashion. It has the benefit of addressing delocalisation and encouraging a deeper understanding of the code. However, progress through code documents can be slow, and the systematic approach enforces a particular strategy and reading order, which consequently, can lead to some apparently simple defects being missed.

To balance the systematic reading technique and combat the potential flaws, a checklist-based approach was selected. The checklist approach aims to address defects missed by the systematic technique's linear strategy. This is one of the more traditional inspection techniques that are widely used in industry [27], [36], [73]. Using the checklist technique also allows for a comparison between the effectiveness of a traditional technique with techniques that have been developed to deal with the specifics of object-oriented code.

From the results of the previous experiment, it emerged that addressing the dynamic aspects of object-oriented code may be beneficial for inspection. In response to this, a use-case driven reading strategy was developed as the third reading technique. Use-cases form part of the Unified Modelling Language (UML) and more information on use-cases may be found in [15], [75], [89]. The remainder of this section presents a description of each of the three reading techniques.

5.1.1 Checklist

Checklists are a straightforward and commonly used reading support mechanism (they have been around since the early use of inspections in the late 70's) used by individual inspectors for the purpose of preparation. Checklists are based upon a series of specific questions that are intended to focus the inspector's attention towards common sources of defects. Gilb and Graham [36] and Humphrey [40] recommend that checklists should not be composed of general, potentially irrelevant questions obtained from elsewhere.

Laitenberger *et al.* [50] summarised a list of the weaknesses of the checklist technique. Firstly, the questions are often general and not sufficiently tailored to a particular development environment. Secondly, concrete instructions on how to use a checklist are often missing, i.e. it is often unclear when and based on what information an inspector is to answer a particular checklist question. Finally, the questions of a checklist are often limited to the detection of defects that belong to particular defect types. Since the defect types are based on past information [19], inspectors may miss whole classes of previously undiscovered defects (a problem that the recommended constant revision of checklists attempts to address).

To overcome the first checklist problem concerning general and unrelated questions, the questions in the checklist were based on historical defect data. The historical information came from the two previous controlled experiments investigating the inspection of object-oriented code (presented in Chapter 3 and Chapter 4).

Combining the defects from the previous two empirical studies created a list of forty-six defects (of which nineteen contained delocalised features). For each of the defects a series of specific questions were derived that should have helped an inspector find that defect.

Gilb and Graham [36], state that a checklist does not need to contain every single question, and should instead concentrate on questions which will turn up major defects and all of which fit onto one page (approx. 25 items). This limit is also agreed upon by Chernak [19], although there are some checklists that do not always adhere to this [2], [66]. The questions were then reviewed, and in some cases merged or generalised as they covered similar areas, to produce a final list of eighteen questions.

The format of the checklist follows that used by Laitenberger *et al*. [51] and suggested by Chernak [19]. It consists of two components, “where to look” and “how to detect”. The first component is a list of potential “problem spots” that may appear in the work product, and the second component is a list of hints on how to identify a defect in the case of each problem spot. This provides more concrete instructions on how to use the checklist. The eighteen derived questions were reviewed and grouped by the area of code they focused on, e.g. inheritance, data referencing, and method overriding.

A final step applied to the construction of the checklist was ordering the questions to support the inspector in building up a thorough understanding of the code and minimise context switches.

As the inspector moves through the different groups of questions (e.g. method, object messaging) they successively move from a more high level and general perspective, towards a more detailed and fine-grained perspective. Each group of questions requires more and more understanding of each method, and so the final question in the method section, “Does the method match the specification?” should be easier to answer once all the other questions have been applied. To support this strategy further, interdependencies (degrees of coupling) within the code under inspection were analysed and those classes with least dependencies were inspected first.

For each class:		
	Feature	***Question***
1	**Inheritance**	Is all inheritance required by the design implemented in the class?
2		Is the inheritance appropriate?
3	**Class Constructor**	Are all instance variables initialised with meaningful values?
4		If a call to `super` is required in the constructor, is it present?
For each method:		
5	**Data Referencing**	Are all parameters used within a method?
6		Are the correct class constants used?
7		Are indices of data structures (arrays, etc.) operating within the correct boundaries?
8	**Object Messaging**	Is the correct method being called on the correct object (including the possibility of casting)?
9		Are the correct values passed as parameters in the correct order?
10	**Object Referencing**	Should a reference to an object be used instead of a distinct copy (or vice versa)?
11	**Selection and Iteration (if, while, etc)**	Are all relational and logical operators sufficient and correct?
12		Is the correct sequence of code executed for any condition outcome?
13		Is the use of an iterator or loop appropriate when destructive operations are occurring on a collection?
14	**Method Behaviour**	Are all assignments and state changes made correctly?
15		For each return statement, is the value returned and its type correct?
16		Does the method match the specification?
For each class:		
17	**Method Overriding**	If inherited methods need to behave differently, are they overridden?
18		Are all uses of method overriding correct?

Figure 5.1 – Final version of derived checklist

This principle was also applied to the “where to look” component and the questions were categorised into three sections:

1. Class – this section is concerned with inheritance and constructor issues.
2. Method – the middle section of questions deals with issues surrounding methods, e.g. data referencing, object messaging and referencing, selection and iteration, and method behaviour.

3. Class – the final section deals with issues surrounding method overriding – these final class questions appear at the end of the checklist since the answers should be easier to find with an understanding of all the methods in the class.

Humphrey [40] commented that when using checklists, inspectors might find themselves jumping back and forth through the code (as if following method calls). If this happens, the mental context that is created as one method is read will be lost once the inspector switches to reading another. Context switching takes time and often causes errors, increasing the likelihood of a low defect detection rate. When programs are even moderately complex, it is better to review each separate part as a unit. Humphrey suggested that, to reduce the amount of unnecessary context switching, inspectors should complete the entire checklist for each part (method) before they proceed to the next. In the instructions provided to inspectors (found in Appendix D.3.4.2), they are told to apply the method section of the checklist to each method under inspection in turn.

The final version of the derived checklist is shown in Figure 5.1.

5.1.2 Use-case

The use-case reading technique attempts to support inspection of the dynamic execution of object-oriented systems. The aim of the technique is to check that each object is capable of responding correctly to all the possible ways in which it might be used. In other words, is it a good citizen of the system? More precisely, with respect to the use-cases in which the object participates, it seeks to verify that:

- The correct methods are being called.
- The decisions and state changes made within each method are correct and consistent.

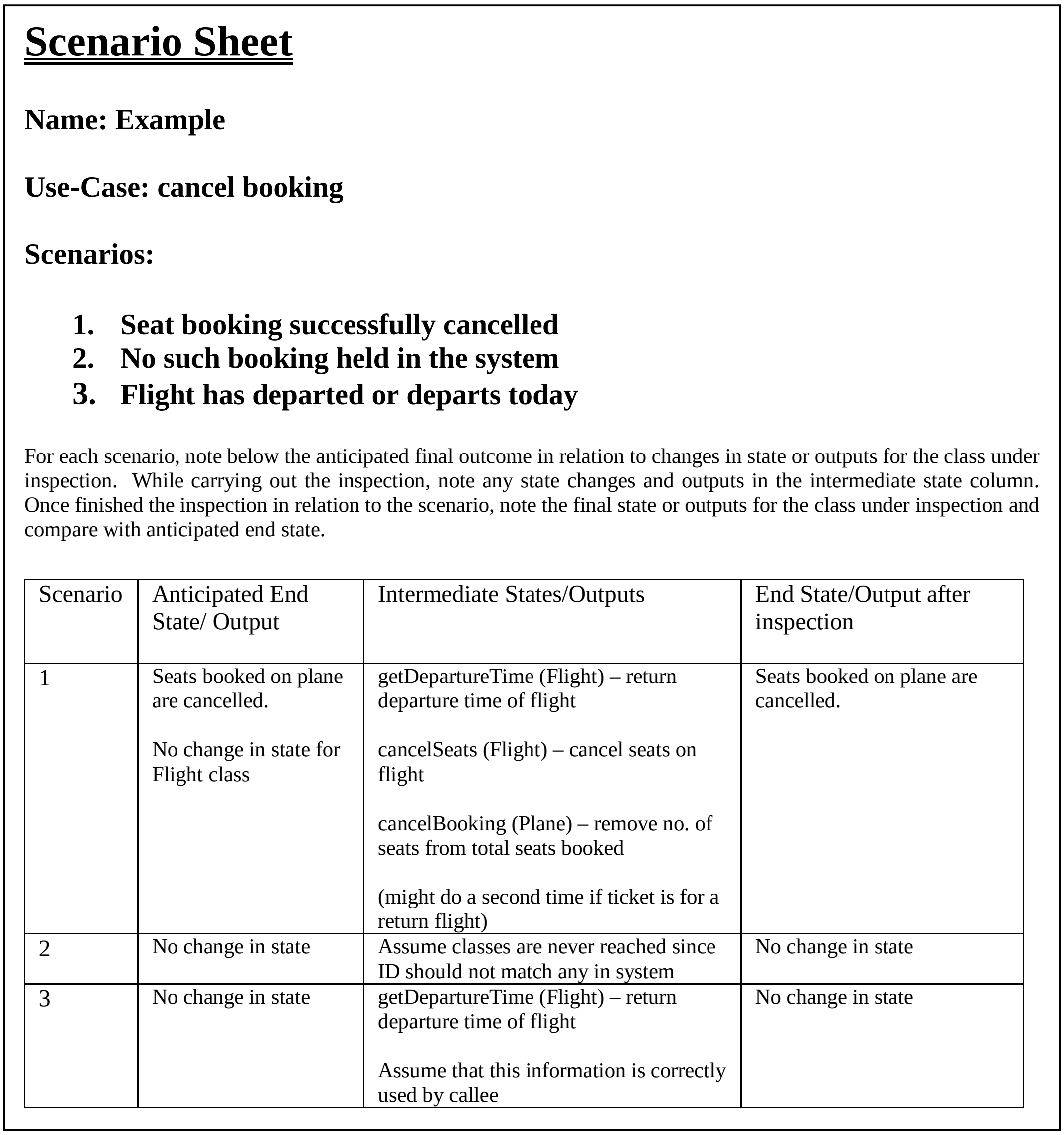
The technique also has the benefit of being an explicit check of the code against the requirements.

The basic approach is to devise a number of scenarios from the use-case and examine how the class under inspection deals with these scenarios. Scenarios are particular instances of a use-case that test possible variations. The principle behind the technique is that it forces the inspector to consider the context in which an object is used. This approach is likely to highlight defects associated with missing or incorrect method calls or erroneous state changes. These are aspects that may be missed if the class was examined in a more general context, e.g. with a checklist. On the other hand, a potential weakness is that some parts of a class may go unchecked because they do not participate in the use-case that is driving the current inspection. Due to this, the approach should be complemented by other reading techniques to ensure complete coverage of a class.

The following briefly describes the steps that should be followed by inspectors when applying the use-case technique.

Creating the scenarios:

- The inspector should take each use-case in turn and devise a series of brief scenarios based on the preconditions, success and failure conditions, and the exceptions described in the use-case. For each scenario the anticipated final outcome in relation to changes in state or output should be noted (see Figure 5.2).



Scenario Sheet

Name: Example

Use-Case: cancel booking

Scenarios:

1. **Seat booking successfully cancelled**
2. **No such booking held in the system**
3. **Flight has departed or departs today**

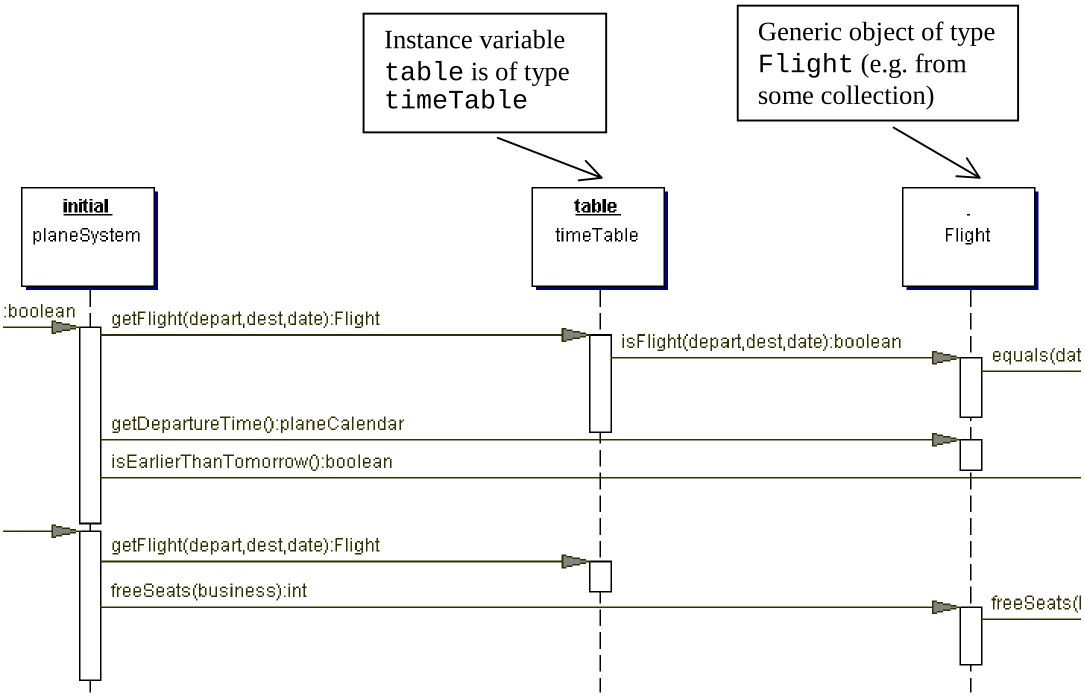
For each scenario, note below the anticipated final outcome in relation to changes in state or outputs for the class under inspection. While carrying out the inspection, note any state changes and outputs in the intermediate state column. Once finished the inspection in relation to the scenario, note the final state or outputs for the class under inspection and compare with anticipated end state.

Scenario	Anticipated End State/ Output	Intermediate States/Outputs	End State/Output after inspection
1	Seats booked on plane are cancelled. No change in state for Flight class	getDepartureTime (Flight) – return departure time of flight cancelSeats (Flight) – cancel seats on flight cancelBooking (Plane) – remove no. of seats from total seats booked (might do a second time if ticket is for a return flight)	Seats booked on plane are cancelled.
2	No change in state	Assume classes are never reached since ID should not match any in system	No change in state
3	No change in state	getDepartureTime (Flight) – return departure time of flight Assume that this information is correctly used by callee	No change in state

Figure 5.2 – Example of a use-case scenario sheet

Using the scenarios:

- The scenario should be traced on the sequence diagram by following the message calls between objects.
- On encountering the class under inspection, the inspector should switch their attention from the sequence diagram to the code, having verified that the expected methods are being called to support the scenario.
- When inspecting the method code any decisions and state changes made should be verified to make sure they are correct and consistent with respect to the scenario. Any intermediate state changes and outputs should be noted (see Figure 5.2). Any method calls made should be followed to verify that they are the correct ones.
- When inspecting a method any method calls made should be followed to verify that they are the correct ones.
  - If the method called is in the class under inspection, the call should be followed and the method code read, otherwise the sequence diagram should be followed.
- Having walked through a scenario, the final state should be compared with the one anticipated and any differences noted as a defect.


Instance variable `table` is of type `timeTable`

Generic object of type `Flight` (e.g. from some collection)

initial
planeSystem
table
timeTable
Flight
:boolean
getFlight(depart,dest,date):Flight
isFlight(depart,dest,date):boolean
equals(dat
getDepartureTime():planeCalendar
isEarlierThanTomorrow():boolean
getFlight(depart,dest,date):Flight
freeSeats(business):int
freeSeats(l

Figure 5.3 – Example of sequence diagram notation used

Figure 5.3 shows an example of the sequence diagrams used. Other features of the sequence diagrams used included:

- Sequence messages detail the name of the method called, the name of any information passed as a parameter, and the type of any return value.
- Sequence message parameters do not show type and are just names to represent the information being passed.
- Complete sequence calls are only shown for the first occurrence, e.g. as shown for the `getFlight` method call shown in Figure 5.3.
- Generic objects have no name (only a ‘-‘) showing only a type. These objects generally occur when an item is accessed in a collection of some sort. Objects with names are instance variables.

The use-case technique, unlike the systematic technique described in the previous chapter, assumes that, as part of the design process, certain material is generated, e.g. a collection of use-cases and sequence diagrams. The sequence diagrams should not be reverse engineered from the code as it may contain defects which are then transferred to the sequence diagrams.

The remainder of this section presents a brief example showing the processes and concepts involved with the technique.

Given the cancel booking use-case shown in Figure 5.4, the following set of scenarios should be derived:

1. Seat booking successfully cancelled
2. No such booking held in the system
3. Flight has departed or departs today

Primary Actor:	Customer
Goal:	Cancel seat booking previously made.
Preconditions:	Person has already booked seat(s) and flight must leave tomorrow at the earliest.
Success Condition:	Seat booking is successfully cancelled and 50% refund on cost is made.
Failure Condition:	-
Trigger:	Customer asks to cancel booking.
Notes:	Information returned to operator (credit card no. and amount to refund) and is dealt with off-line.
Exceptions:	Booking could not be found or flight date is earlier than tomorrow.
Steps:	1. Get booking reference(s) to be cancelled from customer 2. Cancel bookings 3. Make 50% refunds

Figure 5.4 – Cancel Booking use-case

In this particular example the class being inspected is the `planeCalender` class. The anticipated state changes or outputs in relation to the developed scenarios are as follows:

1. No state changes, method should return false
2. No interaction expected
3. No state changes, method should return true

Next, the inspector should follow through the sequence diagram (shown in Figure 5.6), keeping in mind the state of the system (repeating this step individually for each of the derived scenarios). Once a method in the class under inspection is reached (in this case `isEarlierThanTomorrow()` shown in Figure 5.5), the inspector should switch to the code and inspect it, making any notes on changes in state or return values. Finally, once all methods have been reviewed and the sequence diagram has been completely worked through, the inspector should note in the scenario sheet the final state of the class and then verify whether the actual outcomes/state changes match those anticipated at the start.

```
public boolean isEarlierThanTomorrow()
  {
    planeCalendar today = new planeCalendar();
    if(this.get(Calendar.YEAR) == today.get(Calendar.YEAR) &
       this.get(Calendar.MONTH) == today.get(Calendar.MONTH) &
       this.get(Calendar.DATE) == today.get(Calendar.DATE) )
      return true;
    else
      return false;
  }
```

Figure 5.5 – `isEarlierThanTomorrow()`method code

5.1.3 Systematic

The basic systematic technique and its strategy were not significantly altered for use in this experiment (originally presented in Chapter 4.1). Minor adjustments were made based upon feedback and observations from its first usage. Instructions given to subjects were made clearer and more specific (via an instruction sheet provided to subjects during the exercise), and more training and examples were provided for subjects. The amount of information subjects had to write on the abstraction sheets was reduced to help speed up the process. Subjects no longer had to list inherited methods and instance variables. This information was provided for inspectors, since it could be auto-generated prior to the inspection.

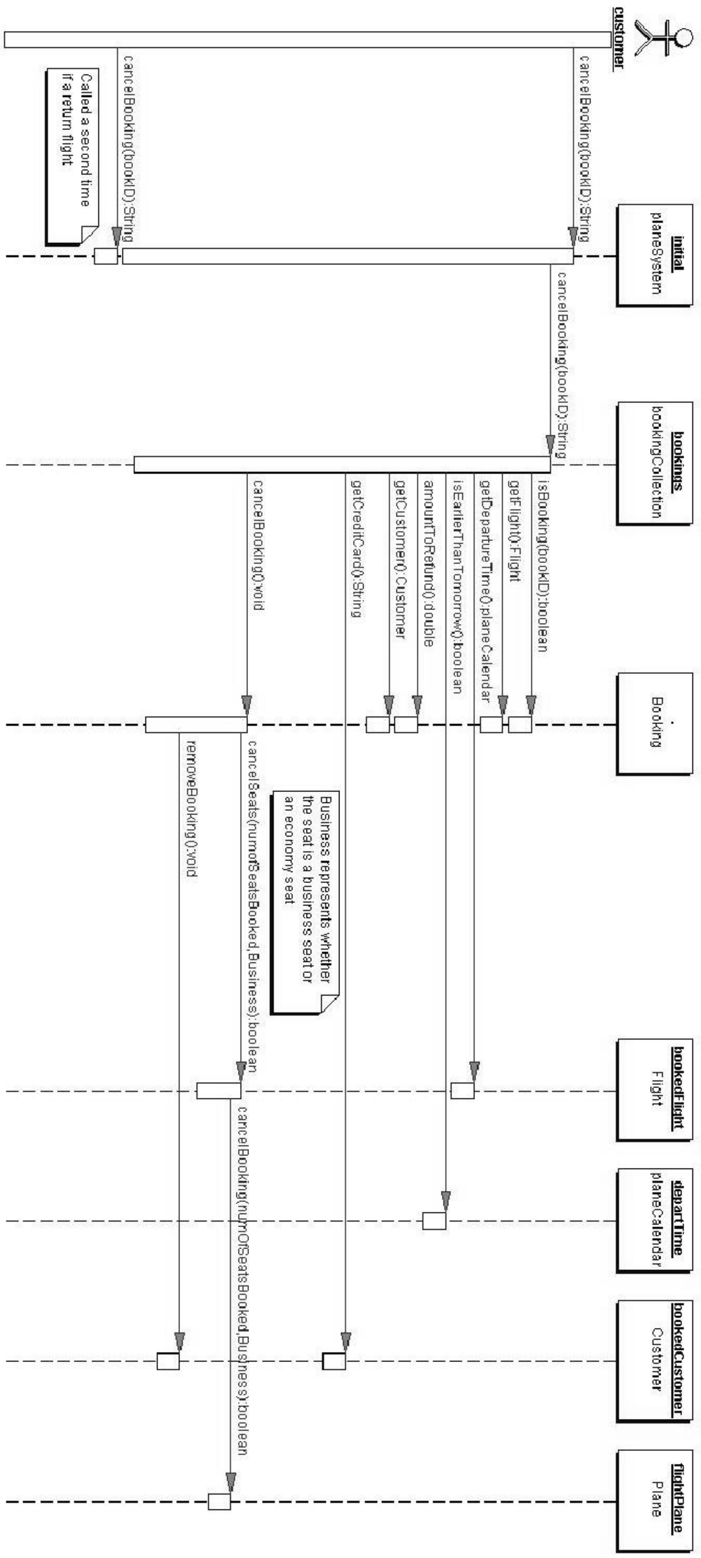

customer
initial
planeSystem
bookings
bookingCollection
:Booking
bookedFlight
Flight
departTime
planeCalendar
bookedCustomer
Customer
flightPlane
Plane
cancelBooking(bookID):String
cancelBooking(bookID):String
Called a second time if a return flight
cancelBooking(bookID):String
isBooking(bookID):boolean
getFlight():Flight
getDepartureTime():planeCalendar
isEarlierThanTomorrow():boolean
amountToRefund():double
getCustomer():Customer
getCreditCard():String
cancelBooking():void
cancelSeats(numofSeatsBooked,Business):boolean
Business represents whether the seat is a business seat or an economy seat
removeBooking():void
cancelBooking(numOfSeatsBooked,Business):boolean

Figure 5.6 – Sequence diagram for Cancel Booking use-case

5.2 Empirical Evaluation

5.2.1 Introduction

To compare the three reading techniques, a controlled experiment was devised to evaluate them primarily in terms of defects detected, but also to consider factors such as efficiency and usability. A copy of all the material used for the actual experiment, including details of the defects used can be found in Appendix D.

5.2.2 Experimental Goals and Hypotheses

The aims of the experiment were again focused using the Goal Question Metric (GQM) paradigm as described by Solingen and Berghout [87].

Goal 1

Analyse the effectiveness of the checklist, systematic, and use-case reading techniques **for the purpose of** comparison **with respect to** their detection of defects **from the viewpoint of** a researcher **in the context of** a University lab course using Java.

This is the main goal of the experiment, evaluating the three reading techniques as an aid for defect detection during inspection of object-oriented code. To meet this goal requires answering the following question:

Q1.1: Is there any difference in the number of defects found by either the checklist, systematic, or use-case based inspection?

This question may be answered by collecting data for the following metrics:

M1.1.1 Number of defects found, classified by inspection technique

Testable hypotheses are derived from the statement of goals, the questions and the metrics as follows:

H1: The null hypothesis, H_0, for the experiment can be described as:

> There is no significant difference between the number of defects found by those subjects performing checklist, systematic or use-case based inspection of object-oriented code.

The alternative hypothesis, H_1, is:

> There is a significant difference between the number of defects found by those subjects performing checklist, systematic or use-case based inspection of object-oriented code.

Goal 2

This second goal of the experiment is more exploratory in nature and is aimed at investigating the affect of the delocalised defects on the different reading techniques, as well as looking at the different types of defect found by each technique.

Since the second goal is exploratory and relies on a qualitative analysis, no testable hypotheses are derived.

5.2.3 Experimental Plan

The experiment used a between subjects design, with three groups of twenty-three students of approximately equal ability based upon marks from previous classes (see Table 5.1). Each group was assigned just one of the reading techniques. This choice of design was made for practical reasons. The experiment was to be carried out within in a third year software engineering course and had to fit within the time constraints of this class. The drawback of this approach as compared with a 3x3 factorial design is that fewer data points would be available, but it had the advantage that ordering effects (due to using different reading techniques) did not have to be dealt with.

In response to a weakness identified in the design of the previous experiment, subjects were given two weeks of education and practice in their assigned reading technique. This consisted of a one-hour lecture on that group’s technique and two laboratory sessions where they were able to practice using the technique and ask any questions. In the second week of practice, the group phase of the inspection process was introduced. This group phase was carried out by creating groups composed of all three reading techniques and asking them to create a final defect list through the usual process of document reading and discussion. The group phase served two purposes. It allowed subjects to form opinions on the other reading techniques and their effectiveness (this was necessary for a report they were required to write-up individually after the experiment - see Data Collection section), and secondly, it gave students a more complete experience in inspections. Before the group phase began all individuals experimental data was copied to maintain its integrity. Since the focus of the experiment was the performance of the individual inspector, the group results have been omitted from the formal analysis (more on this can be found in [28]).

	Reading technique	Number of subjects
Group A	Checklist	23
Group B	Systematic	23
Group C	Use-case	23

Table 5.1 – Inspection order

The experiment proper was carried out in the third week and lasted for one afternoon, consisting of the individual phase lasting ninety minutes, followed by the group phase (with a short break in between). The code inspections carried out were paper-based, with some material available via a web browser (e.g. sequence diagrams, use-cases, class specifications). No tool support was provided.

Subjects

Subjects were participants in a 3rd year Honours Computer Science Software Engineering course run at Strathclyde University. 69 subjects were participating in the class. Subjects had previous experience with the programming languages of Java (two out of twelve first year credits and three out of twelve second year ones) and C (one out of twelve second year credits). The subjects had limited knowledge of Software Requirements Specification (SRS) document inspection, and no experience with code inspections. It should be noted that these subjects were a completely different set from the previous two experiments.

Prior to the experiment, subjects were given a problem statement describing an airline booking system (the original problem statement can be found in Appendix D.1). From this initial specification, subjects were given six weeks to derive a specification for the system. Once completed, subjects were then provided with a specification prepared by the course lecturer. From this subjects were given a further six weeks to code the airline booking system using Java. It was after this stage in the course that the experiment took place.

Statistical Power

Statistical power analysis is a method that can be used to increase the probability that an effect has been found in an experiment (more information regarding statistical power can be found in Miller *et al*. [64] and Welkowitz *et al*. [95]). It reduces the chances of falsely rejecting the null hypothesis or falsely accepting the null hypothesis. If a test does not have sufficient statistical power, then the experiment may not have enough information to allow any reliable conclusions to be made using statistical significance testing. The effect size represents the degree to which the phenomenon under study is present in the population. The larger the effect size, the greater the probability the effect will be detected, and the null hypothesis rejected. Unfortunately, the results from the previous experiment did not have any conclusive results, therefore a large effect size cannot be assumed for this experiment.

Based on the recommendations by Miller *et al*. [64] and because of the inconclusive results from the previous experiment, a medium effect size of 0.5 will be assumed. In this experiment, the hypothesis is assumed to be two-tailed, since the direction of the result is not known ($\propto = 0.05$). The sample size or the harmonic mean for this experiment was derived to be 23. From this, the power of this experiment was found to be 0.4. A potential issue with this experiment is the lack of power. With an approximate power level of 0.5 and assuming a medium effect size (0.5), the number of subjects required in each group would have to be 31 to be able to obtain a significant result. This experiment only had 23 subjects using each technique. For this number of subjects to be acceptable, the effect size would have to have been 0.59. It should be noted that a significant result could still be obtained with fewer subjects, but that the chances of falsely accepting or rejecting the null hypothesis will increase. This has to be kept in mind when considering the results.

Code and Defects

Java was used again because the experiment required an object-oriented language and the subjects had been using the language for the proceeding 2.5 years. As with the previous experiments, the code document used for inspection were approximately 200 lines in length, to be inspected in 90 minutes. The amount of code inspected is in line with established practice (see Chapter 3.2.3).

For the practice sessions of the experiment, subjects were presented with material taken from a sample solution prepared for the airline booking system. For the recorded session of the experiment, the material used represented an extension to the airline booking system which allowed for reservations to be made. The extension consisted of two Java classes. Subjects had not previously seen any code documents or specifications for this extension.

The defects were based on several sources; two previous experiments investigating object-oriented inspections presented in Chapter 3 and Chapter 4, information collated from the literature (Chapter 2), an industrial survey (Chapter 3), and a selection of naturally occurring defects (i.e. appeared in the code when written by the author). In total fourteen different defects were seeded into the code document. Since the experiment was interested in investigating the effects of delocalised defects, eight of the defects seeded had delocalised features.

Paper and Web material

For each individual inspection, subjects were presented with a booklet containing the relevant material (inspection and reading technique instructions, support material such as checklists, scenario sheets or abstraction sheets, code listings, and defect report forms). As well as the paper based material provided for the inspection, extra material was made available to inspectors via a local web page. This page differed depending on the technique used and is summarised in Table 5.2. All code made available via web pages was in plain text and contained no special highlighting, comments or hypertext links.

Technique	Material Available
Checklist	Class diagram Specifications for all classes in system Any code previously inspected
Systematic	Class diagram Specifications for all classes in system Abstractions for other system classes that would have already been inspected had the overall strategy of inspecting those classes with least dependencies first been followed Any code previously inspected
Use-case	Class diagram Specifications for all classes in system Use-cases to be inspected Sequence diagrams for use-cases Any code previously inspected

Table 5.2 – Web material made available during inspections

Data Collection

For all inspection techniques, inspectors were provided with a defect report form on which to record defects found. For systematic inspections, inspectors were given method specification sheets. These contained boxes in which subjects were to write their abstract specifications for each method. Use-case inspectors were provided with a scenario sheet on which to record their derived scenarios for a specific use-case, as well as the anticipated, intermediate and final state changes and return values.

After the inspection exercise was complete, subjects were given a week to write a report on the individual phase of the inspection.

They were asked to include:

- A comparison and analysis of the defects discovered by the different techniques.
- A description of the way they applied their technique (including any deviations made and problems encountered).
- A consideration of the defects their technique failed to discover.
- General comments about the strengths, weaknesses and possible improvements for their technique.

The primary purpose of the report was for assessment purposes, but they were also scrutinised for any insightful comments on the techniques.

Data Analysis

The goals of the experiment feature both a testable hypothesis and exploratory analysis. Since there was one independent variable (the reading technique), three experimental conditions (i.e. the three reading techniques used by subjects), different subjects within the three reading technique groups, and the data was non-parametric in nature, the Kruskal-Wallis test was used (using SPSS) to determine whether the defect results for the three techniques were significantly different.

The remaining goal that was exploratory in nature was investigated through the analysis of the qualitative information gathered during the experiment and from the post experiment subject reports.

Threats to Validity

The potential threats to the internal and external validity of the experiment were the same as those for the previous experiments.

5.2.4 Experimental Procedures

The following timetable was used:

Week 1: Introductory lecture on inspection.

Week 2: Lectures and practice for each of the three individual reading techniques. Lectures lasted approximately 50 minutes and subjects attended a different lecture depending on their assigned reading technique. The practice consisted of a

90 minute training session the following day. The training session was run informally to allow subjects to ask questions and to overcome any conceptual problems about the inspection process and the technique they were using.

Week 3: Lecture for group activity and practice with individual and group inspection. The practice session consisted of a 60-minute individual inspection task (with subjects using their assigned technique) and was followed by a 45-minute group activity task. The session was again run informally to help subjects overcome any problems encountered.

Week 4: Inspection experiment proper (individual inspection followed by group inspection). Subjects were given up to a maximum of 90 minutes to complete the inspection. Once subjects had finished the inspection task, they were allowed a 10-minute break before forming into their groups for the group activity. Subjects were given up to 45 minutes to complete this part of the experiment. The individual inspection task was completed under exam conditions.

5.2.5 Experimental Results and Analysis

The results are based upon the 69 subjects who participated. The following sections describe the results of the various elements of the inspection experiment.

		Inspection Technique		
		Checklist	**Systematic**	**Use-case**
Number of subjects:		23	23	23
Defects (out of 14):	Mean	7.3043	6.1739	5.7391
	Std. Deviation	2.4943	2.2290	2.3973
	Std. Error	.5201	.4648	.4999
	Minimum	2	3	2
	Maximum	11	10	10
False Positives:	Mean	3.4348	3.2174	2.8696
	Std. Deviation	2.6939	2.8116	1.9841
	Std. Error	.5617	.5863	.4137
	Minimum	0	0	0
	Maximum	12	10	7
Inspection Time:	Mean	72.1739	77.0000	81.9130
	Std. Deviation	12.9568	9.7933	9.2830

Table 5.3 – Summary of results for third experiment

Defect Detection (Individual Performance)

The main results of the experiment are contained within Table 5.3. The results show that the checklist reading technique subjects were finding the most defects, compared to the systematic and use-case subjects. Both the standard deviation and standard error results for the number of defects found for all the techniques are within a similar range, with no technique showing erratic results (which might have suggested a problem either with the subject partitioning or one of the reading techniques). The maximum and minimum number of defects found by subjects using each technique were also similar. This indicates that no one technique was significantly superior or inferior to any of the others – some subjects performed badly, other performed well, no matter the technique.

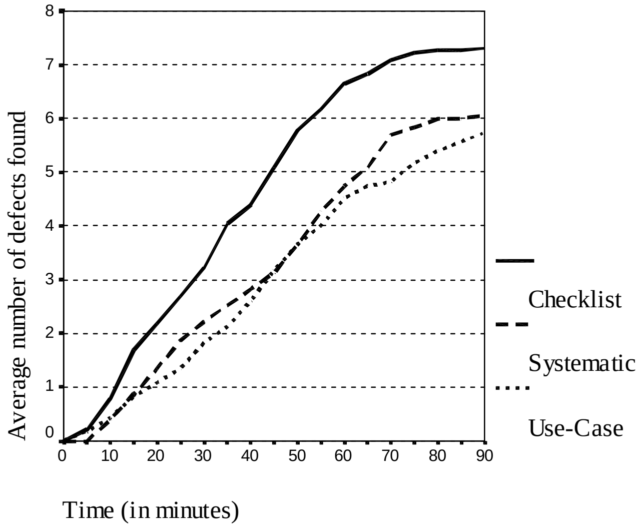


Figure 5.7 – Defect response rates

False positives are defects noted during an inspection which turn out not to be defects. The results show that out of the three reading techniques, subjects using the use-case technique were writing down the least number of false positives. This is reflected in the mean, standard deviation, error and maximum values. In comparison, the checklist technique had the highest number of false positives. One possible reason for this may be related to the level of understanding enforced by the technique. The checklist does not particularly encourage understanding, whereas the systematic and use-case techniques both require understanding or a mental execution of the code, perhaps leading to a reduction in the creation of false positives. Alternatively, the reason for the difference may be due to technique overhead. The checklist had the lowest technique overhead, perhaps allowing subjects to spend more time re-reading the code and identifying false positives.

The defect detection rates for each of the three reading techniques are shown in Figure 5.7. The x-axis shows the time during the inspection, the y-axis shows the average number of defects found, and the three lines in the graph represent each of the reading techniques.

Subjects using the checklist technique find more defects and at a quicker rate, although performance levels drop off sharply after the first 60 minutes. The defect detection rates of the systematic and use-case subjects appear to be fairly similar, with systematic subjects' performance levelling off towards the end of the 90 minutes. The use-case subjects' performance does not appear to be levelling off in the same way. This may suggest that defects were still being found at the end of the 90 minutes. Subjects using the checklist technique appeared to find defects quicker than those using the other reading techniques. This may be because the checklist does not have the technique overheads of the other two (e.g. writing abstractions or developing scenarios).

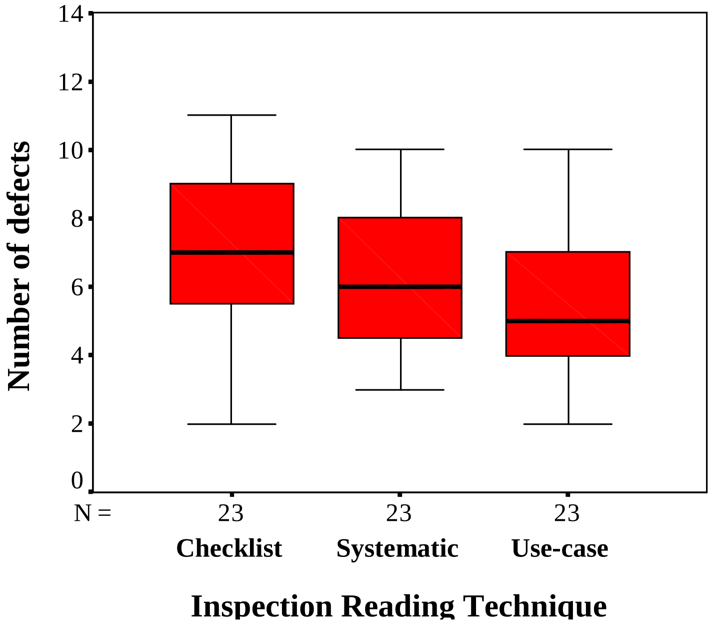


Figure 5.8 – Overall defect detection performance of each reading technique

For this experiment there was one independent variable (the reading technique), three experimental conditions (i.e. the three reading techniques used by subjects), and there were different subjects within the three reading technique groups. Due to the non-normal nature of the results it was not possible to apply parametric statistical tests to determine whether the defect results for the three techniques were significantly different. Instead the Kruskal-Wallis test was used. The results generated by the software package SPSS are shown in

Table 5.4. For 2 degrees of freedom[3], a chi-square result of 4.871 was generated. This results in a significant result at the 10% level (chi-square result > 4.6), but not at the 5% level (chi-square result would have to be > 5.99). The overall defect detection performance of the three reading techniques is shown in the boxplots in Figure 5.8.

In relation to the experiment hypothesis, the null hypothesis, (that there is no significant difference between the number of defects found by those subjects performing checklist, systematic or use-case based inspection of object-oriented code), may be rejected, and the alternate H_1 accepted, but only at the 10% level of significance.

	For 3 techniques
Chi-Square	4.871
Df	2
Asymp. Sig.	0.088

Table 5.4 – Results of Kruskal-Wallis test

Figure 5.9 shows the comparative effectiveness in terms of defect detection for each of the three reading techniques. Those defect numbers along the bottom surrounded by a box are defects with delocalised characteristics. It should be noted that one defect (defect 10) was not found by any inspectors using any of the reading techniques (this was a particularly subtle defect involving the use of a class library). It was also noticed that defects involving some form of omission appear difficult to find (defects 6, 13 and 14).

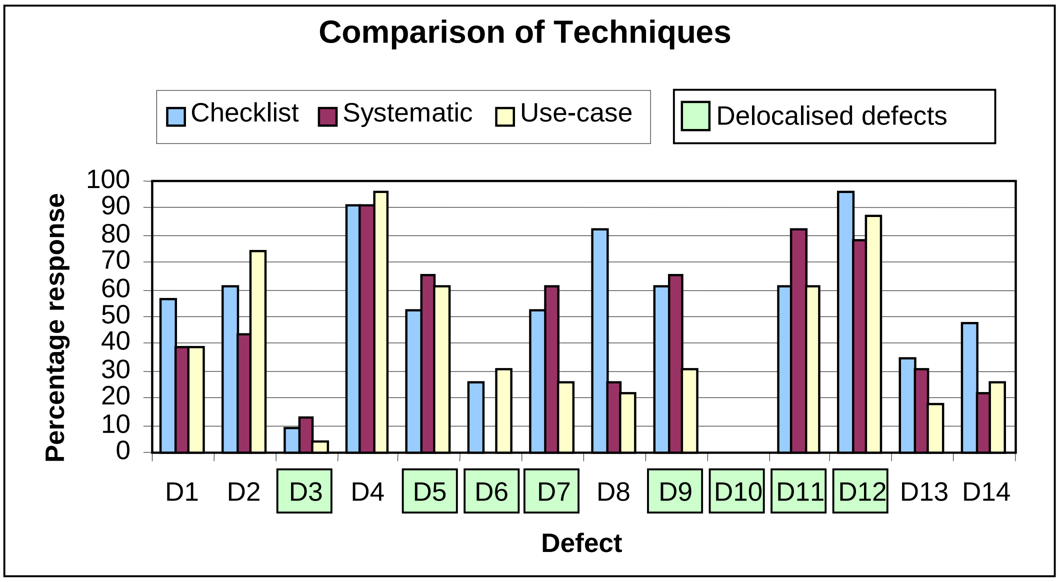


Figure 5.9 – Average technique effectiveness per defect

[3] Degrees of freedom = number of experimental conditions – 1 = 3 – 1 = 2.

All the reading techniques have strong points, and there is not one dominant technique, although the checklist technique performs consistently well. It is noticeable that except in the cases of defect 3, 6, and 10 (which all techniques found elusive), the systematic technique performs consistently well in terms of detecting defects with delocalised characteristics. This is shown better in the three graphs of Figure 5.10, one for each technique (re-imaging the information shown in Figure 5.9). Both the checklist and use-case techniques have a less regular pattern of delocalised defect detection, perhaps reflecting the less exhaustive and more focused nature of these approaches.

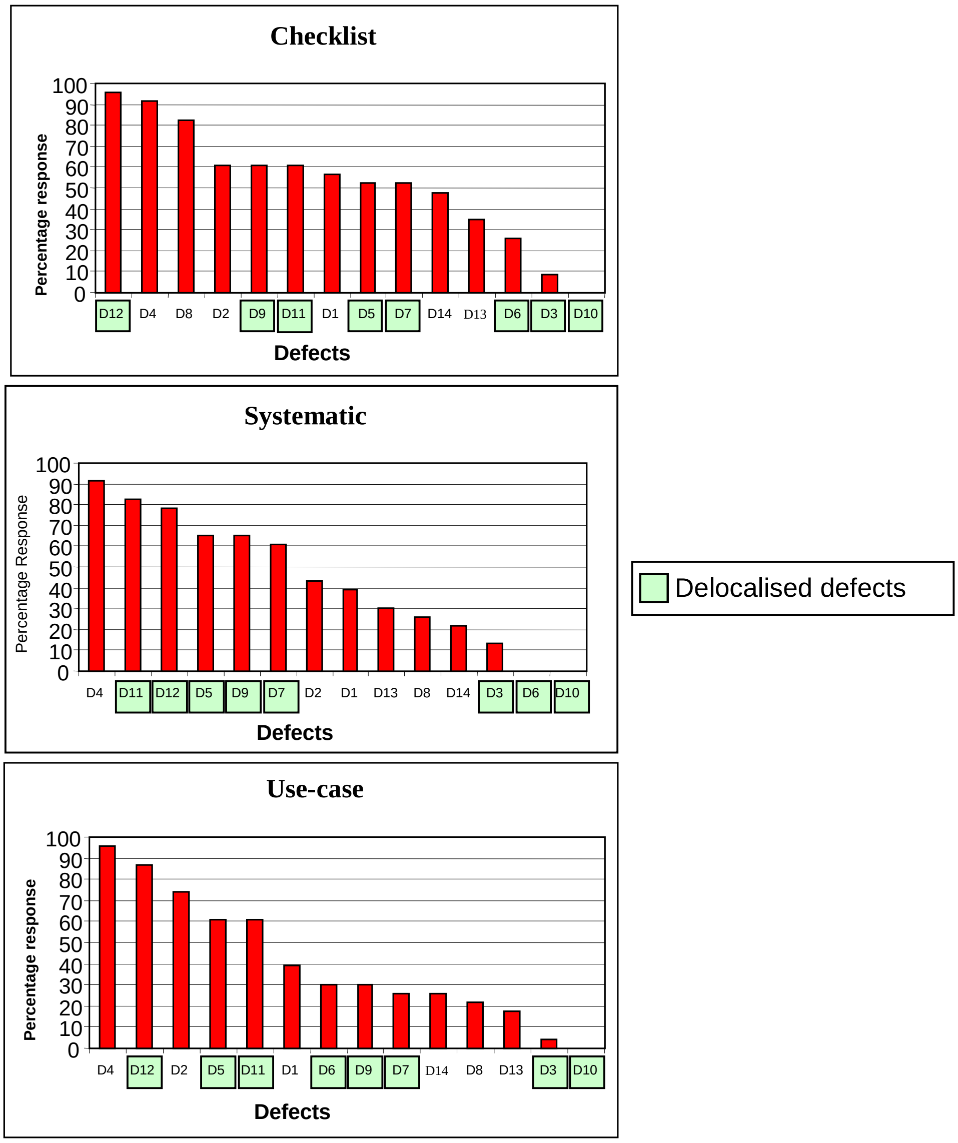


Figure 5.10 – Average effectiveness per defect, split by technique

Figure 5.10 shows that the systematic technique was the only technique to miss more than one defect (defect 6 – consisted of a missing method call). For use-case subjects, defect 6 was highlighted by their sequence diagrams. For checklist subjects, they were encouraged by one of their questions to check the method against the supplied specification (available on the web) – a process that should have highlighted the defect. Systematic inspectors should also have compared their final generated method specification with the online class specification. It may be that there was not enough encouragement for them to do this.

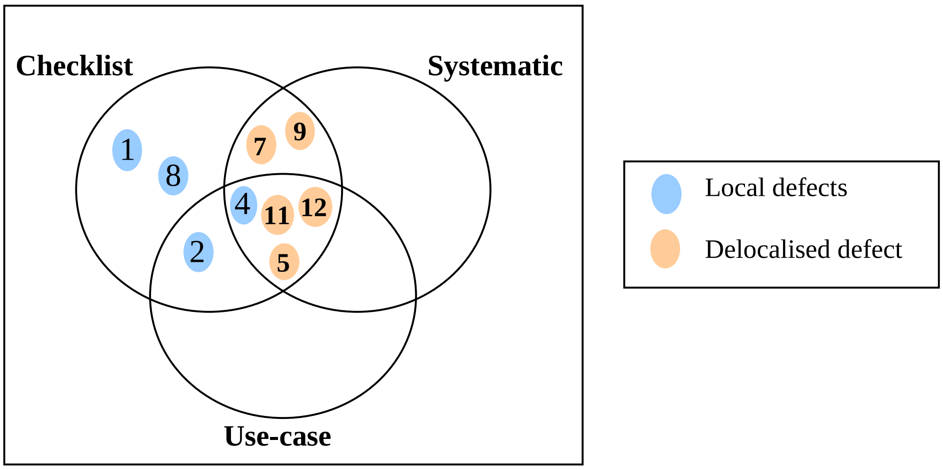


Figure 5.11 – Venn Diagram of defect overlap for techniques (where defects found by more than 50% of subjects)

Figure 5.11 presents a Venn diagram, that shows for each reading technique, the defects found with a detection level greater than 50% (in other words, defects that would have a good chance of being found using a particular technique). This shows that all these defects were found by the checklist technique, while the systematic technique found all the delocalised defects but only one local defect. The use-case technique found a mixture of local and delocalised defects.

One reason for the high response rates for the checklist compared to the other reading techniques may be related to the construction of the checklist questions. The questions for the checklist were generated from defect information gathered from the two previous studies. Although the questions generated were generalised, a threat to the validity of the experiment may be that the defects seeded into the code were too similar to the defects used to generate the checklist. Eleven of the fourteen defects were similar in style to those in the first two experiments. On the other hand, the results can be viewed as being positive for the continued use of checklists, showing that a checklist based on historical information can be effective.

The time that each inspector started and finished his or her inspection was recorded and the results are shown in Figure 5.12. Checklist inspectors were quicker at their inspections and were more likely to have finished before the time limit (indicating a possible reason for the drop in detection rate shown in Figure 5.7). Most of the use-case and systematic inspectors were more inclined to use most, if not all, of the time available. This may have been related to the extra material and thought involved with the technique.

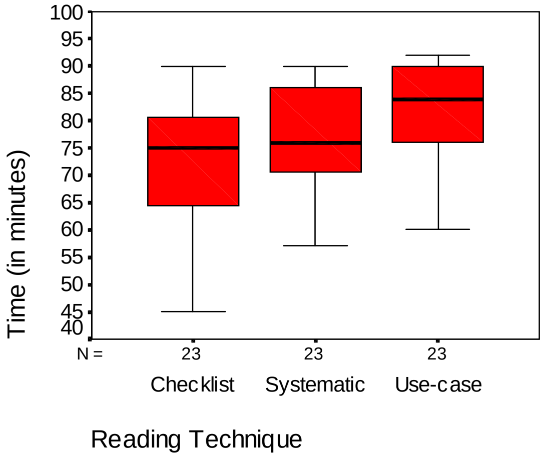


Figure 5.12 – Finishing times for subjects by technique

Defect Analysis

Each of the defects was characterised according to the list of criteria shown in Figure 5.13 (an evolution of the version used in the previous two experiments). The purpose of this was to identify if there was any correlation between defect characteristics and their discovery rate. The characteristics of each defect (columns) ordered by defect discovery rates for each of the three reading techniques (ordered from left to right, starting with the easiest to find defects), are shown in Figure 5.14.

It was observed that all three reading techniques had problems with defects exhibiting the characteristics of wrong object used (D3) and omission (D6, D8, D13, and D14). Defects of omission are very severe and also difficult to detect. Having a checklist question along the lines of “Is all code present?” is not helpful. The question offers no support or guidance on how to identify defects of omission. The systematic approach focuses on understanding what is present, so may only notice omissions when compared with external references (e.g. class or method specifications). Although not evident from the defect results, the use-case technique has the best chance of finding defects of omission as the technique provides an independent source of comparison for the code with software requirements (in the form of scenarios, use-cases and sequence diagrams). Using external sources of comparison appears to offer the best solution to finding defects of omission (although this assumes that the external sources themselves are correct).

Defects involving the use of class library (D7, D10) were found to be difficult for both checklist and use-case techniques, but more evenly spread for the systematic technique.

Defect Descriptors

Use of library class - requires understanding of class libraries

Wrong object used - sending message to wrong object

Wrong method called - sending incorrect message

Incorrect parameter in method call - incorrect parameters in method call

Algorithm/computation - error in the algorithm (e.g. step missing or in wrong order)

Data flow error - incorrect/missing variable or incorrect value

Specification clash - clash with specification

Omission - missing code

Commission - incorrect or superfluous code

Locality - area of code required to be looked at to spot the defect

(M)ethod - information required to identify defect is present at the method level
(C)lass - information required to identify defect is present at the class level
(S)ystem - information required to identify defect is distributed across multiple classes

Method size - size of method where defect present

S = 0-4 lines of code
M = 5-10 lines of code
L = 11 + lines of code

Sequence diagram clash - defect clashes with sequence diagram given to use-case inspectors

Figure 5.13 – Classification scheme for defect characteristics in third experiment

Defects that were delocalised in nature were spread through the range of results for both checklist and use-case responses, but were less spread out and bunched more towards the better response end of the table for the systematic responses. Curiously, for the systematic responses, defects that were not considered to be delocalised in nature (the local defects) were grouped at the lower end of the response scale. It may be the case that the systematic technique helps more with the delocalised defects, but at the expense of the local defects.

Checklist

Defect No.	12	4	8	2	9	11	1	5	7	14	13	6	3	10
Use of class library									X					X
Wrong object used													X	
Wrong method called					X	X	X							
Incorrect parameter in method call	X							X	X					
Algorithm/Computational		X		X						X	X	X		X
Data flow error	X		X	X	X			X		X	X		X	
Specification clash			X	X	X	X				X		X	X	
Omission			X							X	X	X		
Commission	X	X		X	X	X	X	X	X				X	X
Locality (M,C,S)	S	M	M	C	S	S	C	S	S	C	C	S	S	S
Sequence diagram clash	X				X	X		X				X	X	
Method (S, M, L)	M	S	L	M	L	L	M	S	L	M	M	S	M	L
Delocalised	D	L	L	L	D	D	L	D	D	L	L	D	D	D
% response rate	96	91	83	61	61	61	57	52	52	48	35	26	9	0

Systematic

Defect No.	4	11	12	5	9	7	2	1	13	8	14	3	6	10
Use of class library						X								X
Wrong object used												X		
Wrong method called		X			X			X						
Incorrect parameter in method call			X	X		X								
Algorithm/Computational	X						X		X		X		X	X
Data flow error			X	X	X		X		X	X	X	X		
Specification clash		X			X		X			X	X	X	X	
Omission									X	X	X		X	
Commission	X	X	X	X	X	X	X	X				X		X
Locality (M,C,S)	M	S	S	S	S	S	C	C	C	M	C	S	S	S
Sequence diagram clash		X	X	X	X							X	X	
Method (S, M, L)	S	L	M	S	L	L	M	M	M	L	M	M	S	L
Delocalised	L	D	D	D	D	D	L	L	L	L	L	D	D	D
% response rate	91	83	78	65	65	61	44	39	30	26	22	13	0	0

Use-case

Defect No.	4	12	2	5	11	1	6	9	7	14	8	13	3	10
Use of class library									X					X
Wrong object used													X	
Wrong method called					X	X		X						
Incorrect parameter in method call		X		X					X					
Algorithm/Computational	X		X				X			X		X		X
Data flow error		X	X	X				X		X	X	X	X	
Specification clash			X		X		X	X		X	X		X	
Omission							X			X	X	X		
Commission	X	X	X	X	X	X		X	X				X	X
Locality (M,C,S)	M	S	C	S	S	C	S	S	S	C	M	C	S	S
Sequence diagram clash		X		X	X		X	X					X	
Method (S, M, L)	S	M	M	S	L	M	S	L	L	M	L	M	M	L
Delocalised	L	D	L	D	D	L	D	D	D	L	L	L	D	D
% response rate	96	87	74	61	61	39	30	30	26	26	22	17	4	0

Figure 5.14 – Defect characteristics in percentage response order for each reading technique

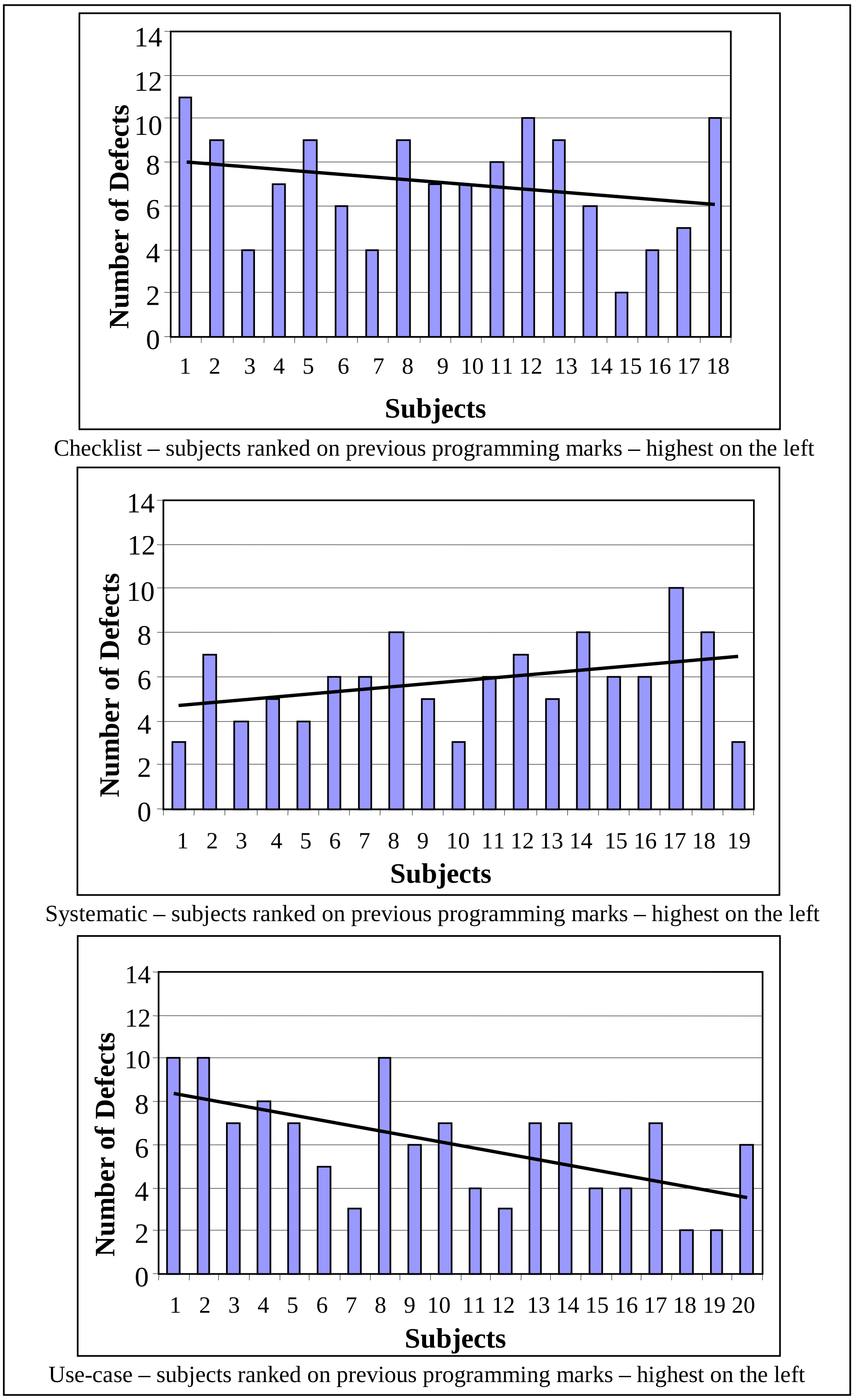
What were unexpected about the results for the use-case technique were the results concerning defects highlighted by the sequence diagram (see Figure 5.14). Those results are spread over the response range (high, middle and low). This may indicate that subjects found it difficult to appreciate the sequence diagrams (due to lack of experience), did not realise that they could be used to compare against the actual code, or were just not using them.

There was one defect predominantly found by the checklist technique compared to both systematic and use-case techniques (defect 8). The defect related to an unused parameter in a method declaration that should have been used. 83% of checklist subjects found defect 8, whereas only 26% of systematic inspectors and 22% of use-case inspectors found the defect. There was a question in the checklist that highlighted this kind of defect, e.g. “Are all parameters used within a method”. Since use-case and systematic inspectors did not have a specific question to answer, they either did not consider an unused parameter a defect or as something that would cause the code to malfunction.

Relationship between Technique and Ability

Figure 5.15 shows, for each of the three reading techniques, the defect detection results of each individual inspector, ordered by their ability levels (based on previous programming class marks) from highest on the left to lowest on the right. The numbers of subjects in these graphs is slightly reduced due to the unavailability in some cases of previous marks on which to base the ranking. Overlapping each of the three graphs is a linear trend line (added using a feature of Microsoft Excel) that shows the general trend of defect detection from the more capable subjects to the less capable. It was expected that for all the techniques, the more able subjects would tend to do slightly better than the less able ones. This was found to be the case for both the checklist and use-case techniques. The trend line for the use-case technique shows a steeper gradient, possible suggesting that the technique was harder to apply.

Notably, the trend line goes in the opposite direction for the systematic technique, suggesting that the more capable subjects were not being as effective as the less capable ones. This was similar to the result found in the first use of the systematic technique in the previous chapter, where it was found that the systematic technique appeared to help the weaker subjects but confound or hinder the natural abilities of the more capable subjects.



Checklist – subjects ranked on previous programming marks – highest on the left

Systematic – subjects ranked on previous programming marks – highest on the left

Use-case – subjects ranked on previous programming marks – highest on the left

Figure 5.15 – Defect detection results of subjects ordered by ability for each reading technique

These graphs demonstrate that although there is evidence for an ability effect, it is not the primary factor when it comes to explaining the defect detecting ability of a technique – the technique itself has significant influence.

Review of Subject Reports

After the inspection exercise was complete, subjects were given a week to write a report on the individual phase of the inspection (more detail on the report can be found in Chapter 5.2.3).

Sixty-eight of the sixty-nine subjects handed in a report (the missing one belonged to the use-case inspection technique). The following summarises the report results, grouped by reading technique.

Checklist (based on 23 reports)

Subjects were asked to discuss how they applied the checklist reading technique in comparison with the guidelines supplied. Five subjects reported no deviation in application of the technique. Other variations included reading the code at the start to get a general view/understanding of the code (six subjects), finding defects then trying to fit them to an appropriate question (five subjects – in doing this they must have been simply reading the code and not applying any specific reading strategy, e.g. ad-hoc), applied one question at a time to the whole class under inspection (five subjects). One subject reported simply reading the code and ignoring the checklist, another read the code at regular intervals without the checklist.

The strengths of the checklist were that it was quick (ten subjects) and straightforward to use (eighteen subjects). Other comments were that it had structure, that it was based on past experiences and that it was less reliant on other forms of documentation.

Weaknesses of the checklist included that it was not good at detecting defects associated with missing lines of code (twelve subjects), that it does not encourage understanding the code being inspected (eight subjects), and that the checklist questions can be too vague (five subjects). Other weaknesses included that it relied too much on the preparation of an appropriate set of questions, is susceptible to human error, does not help with the bigger picture (e.g. the closer you get the less you see), can get tedious and repetitive, and that there is no encouragement to read supplementary documentation.

Possible improvements to the checklist included making the questions more specific and with subsections (five subjects) and forcing the inspector to get a better understanding of the code (four subjects).

Many subjects (eleven) suggested that the failure of the checklist technique to find defects was not the fault of the checklist itself, but due to their relative inexperience. Other suggestions included that the checklist should be used in conjunction with at least one other technique, e.g. systematic technique (three subjects), while several others suggested that the checklist does not deal with object messaging well enough (three subjects).

Systematic (based on 23 reports)

In applying the systematic reading technique (in comparison with the guidelines supplied), fourteen subjects reported no deviation in application of the technique. Variations on the application of the technique included occasionally making assumptions about what the code was doing, not always reading an abstraction if it was available, reading all the code at the start before applying the technique, assuming the function of external methods, and if a defect was spotted easily no comparison was made with the provided class specification.

The main strength of the systematic technique was that it promoted a deep understanding of the code under inspection (eleven subjects). Other strengths were that as a by-product, method specifications were created for later use, that every line of code was looked at, that it was easy to use, and that the systematic technique was good at helping with defects related to incorrect procedure calls.

The main weakness of the systematic technique was that it was not good at detecting defects associated with missing lines of code (seven subjects). Other weaknesses included the difficulty of writing the natural language method specifications, that the technique was too time consuming, following external references was time consuming and produced a cognitive overload, and defects on a global scale are not dealt with well. Two comments were made relating to the class specifications supplied during the inspection. Subjects reported that these specifications were not detailed enough, and that the systematic technique relied too much on comparison with these specifications.

A further weakness concerned the writing of method specifications where defects had already been found. Currently, even if a defect has been found before the method specification has been written, it has to be incorporated into the method specification. The presence of the defect may complicate the description that has to be written, slowing the inspector. It may be that it is not prudent to write the method specification, since at some later stage it would have to be re-written to accommodate the corrected code. But, on the other hand, if the specification was not written, other defects still within the method may be missed.

Possible improvements to the systematic technique included revising the contents of the method specifications to be longer and more detailed, having a more efficient way of tracing external references, recording defects and abstractions electronically, making use of sequence diagrams, or reading less during inspections and concentrating more effort towards non-trivial methods. One other suggested improvement to the systematic technique was to apply aspects of the checklist technique to each method before writing each method specification.

Many subjects (eleven) suggested that the failure of the systematic technique to find defects was not the fault of the systematic technique itself, but due to their relative inexperience. Subjects suggested that the systematic technique complimented the other reading techniques used (two subjects) and that it relied too much on the ability of the programmer (three subjects). Two subjects commented that the documentation available was too basic.

Use-case (based on 22 reports)

Six subjects reported no deviation in application of the technique. Six subjects claimed that they had a general read through the code to spot any obvious defects before they started using the technique. Five subjects stated that they did not write down any state or intermediate state information during their inspection. One subject continually asked themselves lots of questions (which were subsequently found to appear on the checklist) when reading the code. Two other subjects claimed that they stopped using the technique in mid inspection (possibly due to running out if time) and just read the code.

The main strength of the use-case technique was that methods were dealt with within the context of the system executing (five subjects). Other comments included that the technique complimented the object-oriented nature of the code (three subjects), that it was easy to use (two subjects), and that a better understanding of various aspects of the code was achieved (three subjects). One or two subjects responded positively to the opportunity to use sequence diagrams, and suggested that the technique had potential to help highlight missing method calls (dependent on the quality of the sequence diagrams).

The main weaknesses of the use-case reading technique were that it was slow and time consuming (ten subjects), not all the code was covered (eight subjects), and that there was too much jumping around between different documents and diagrams (seven subjects). Other weaknesses included finding the recording of state information annoying (four subjects), that the use-case technique had problems with defects associated with missing code (four subjects), and the technique relied too much on user generated scenarios (two subjects). One subject commented that the inspector could end up generating a large list of intermediate states for complex scenarios. Seven subjects had difficulty with the presentation and readability of the large sequence diagrams used (stating a desire for a paper copy). One subject commented that there was not enough detail in the sequence diagrams.

The main improvement suggested by eight subjects to the use-case technique was the introduction of a checklist in some form. Other improvements included grouping all methods together for a particular sequence diagram (to help reduce jumping around), and to reduce the amount of writing. Several comments concerned the generation of the scenarios, one suggestion being that the scenarios should be pre-produced before the inspection, another suggesting that the scenarios should be generated by at least two people. One subject thought that the sequence diagram and the code should be integrated together. Another suggestion was that the sequence diagrams should be provided in paper form (not ideal since in the preparation for the experiment it was found that many sequence diagrams could become very large and unwieldy on paper). One other comment suggested producing the scenarios and defect lists electronically.

Many subjects (twelve) suggested that the failure of the use-case technique to find defects was not the fault of the use-case technique itself, but due to their relative inexperience. Five subjects suggested that the use-case technique should be used in combination with other techniques. Finally, one subject claimed that the technique was vaguely defined when compared to the checklist technique.

5.2.6 Interpretation of Results

The main result of the experiment is that there is evidence of a significant difference between the number of defects found by those subjects performing checklist, systematic or use-case based inspection of object-oriented code when working at the 10% level of significance. The checklist approach was the most effective reading technique, followed by the systematic approach, which showed signs of dealing with delocalised defects better.

The remainder of this section looks at each of the reading techniques, highlighting the positive and negative aspects, and the potential benefits and weaknesses.

Checklist

Potential benefits of the checklist technique included:

- The checklist was found to be the most effective technique, even when being used by less able subjects.
- Anecdotal evidence from subject reports suggests that the subjects found the technique to be easy and straightforward to use.
- The timing information shows the checklist to be the quickest technique to apply.

There may be some weaknesses surrounding the use of the checklist reading technique:

- It does not deal well with defects related to missing lines of code (this is in common with the other techniques).
- It fails to push inspectors towards a deep understanding of the code under inspection (although in this case it does not seem to have a too detrimental effect).
- The questions have to be phrased in a way that is neither too general nor too specific.

Suggestions for improvements to the checklist by subjects include forcing inspectors to obtain a better understanding of the code and making the questions more specific.

Systematic

Some of the potential benefits of the systematic technique included:

- There was evidence that the systematic technique was effective at dealing with delocalised defects.
- There was anecdotal evidence from the subjects' reports that the systematic technique encouraged a deeper level of understanding of the code under inspection.
- The systematic technique produced abstractions for every method as a by-product of the approach. These abstractions can be used in future inspections to save the inspector, or other inspectors, the effort of reading the class or method again when another class makes a delocalised reference to that class.
- The systematic technique appeared to help the weaker subjects but suppress the defect detection abilities of the more able subjects (a result similar to that found in the first application of the systematic technique in the second experiment).

Weaknesses that could affect the use of the systematic reading technique include:

- It does not deal well with defects related to missing lines of code.
- It relies on the presence of a class specification against which the derived abstractions are compared.
- The detail and content, as well as the full benefit of the generated abstractions have yet to be fully evaluated.
- Currently, even if a defect has been found before the method specification has been written, it has to be incorporated into the method specification. Including the defect may complicate the abstraction and would require the abstraction to be re-written once the defect has been removed. Not writing the specification may mean that other defects within the method could be missed.

Suggestions for improvements to the systematic technique suggested by subjects included:

- Supplying some form of optional checklist to help inspectors verify that they have covered all the important aspects of the code.
- Evaluate the contents of the derived abstract specifications (making sure they are relatively quick and easy to derive, but contain enough information to be useful in the future).

Use-case

The main potential benefit of the use-case technique:

- Encourages inspection of code from a dynamic viewpoint and provides a technique that explicitly compares code against requirements (via scenarios, use-cases, and sequence diagrams).

Although the use-case technique did not perform as well as the checklist and systematic techniques, it has the potential to offer an independent source of comparison for the code against requirements (via use-cases, generated scenarios, and sequence diagrams) to help highlight defects. This may also help with defects of omission.

Weaknesses that could affect the use of the use-case reading technique included:

- The technique was slower and more time consuming than the others.
- Due to the inspection being driven by use-cases, it may be that the technique does not cover all the code.

- Subjects found that there was too much jumping around between the various documents (scenario sheet, sequence-diagram, code).
- Dependent on the creation of good scenarios.
- More subjects using this technique deviated from the recommended application, possibly suggesting a lack of confidence in the technique.

Suggestions for improvement to the use-case technique included:

- Clearer instructions should be given to the inspectors, as well as more substantial examples in the training phase, particularly in the creation of scenarios, which some of the subjects appeared to find difficult.
- Many subject reports suggested incorporating the checklist as part of the process.

5.3 Conclusions

The experiment presented in this chapter explored the use of a systematic, abstraction-driven strategy, a specially created checklist and a use-case driven strategy to address the problems of reading strategy and delocalisation.

The checklist technique was found to be the most efficient and effective of the three reading techniques. The results suggest that if checklists are tailored to the particular development environment using historical defect data and augmented with questions that target object-oriented features then they can be an effective aid for object-oriented code inspection. Subjects also commented that the checklist technique was the least complicated technique and had fewer overheads. However, the usefulness of the checklist relies heavily on the construction of appropriate questions.

The systematic technique provided encouraging results with respect to the detection of delocalised defects. The technique offers a potential long-term advantage through the creation of abstract specifications for each method (but at the cost of a higher technique overhead). Further work is required to determine the long-term benefit of the abstractions in terms of reducing the need to read associated code.

Although the overall results for the use-case technique were relatively weak, it has the benefit of allowing inspectors to read the code from a dynamic model viewpoint. The use-case technique supports a closer examination of inter-class relationships, and through use-cases, generated scenarios, and sequence diagrams, provides a technique that explicitly compares code against requirements. With subjects finding this technique very demanding, further work is required to refine this approach into a practical reading technique.

The next chapter presents a summary of the main contributions and results of this thesis and a set of recommendations concerning the issues surrounding the inspection of object-oriented code.

Chapter 6

Conclusions and Future Work

This thesis has shown that the way in which the object-oriented paradigm distributes related functionality can have a serious impact on code inspection and, to address this problem, it has developed and empirically evaluated three reading techniques.

6.1 Thesis Summary

Although there have been several reading techniques developed to help individual inspectors obtain an understanding of the code under inspection, all were developed at a time when the procedural paradigm was dominant. Object-oriented and procedural languages are different – the encapsulation of data and associated functionality, the common use of inheritance, and the concepts of polymorphism and dynamic binding. These key features may have a significant impact on the ease of understanding of program code and therefore impact upon the effectiveness of inspection.

An investigation of the issues arising from the inspection of object-oriented code found that the characteristics of the 'hard to find' defects included use of class libraries, sending wrong messages to objects, inheritance, overriding and design mismatches. Many of the problem characteristics identified by the investigation were also highlighted by an industrial survey. The key features of object-orientation were found to have a significant impact on the ease of understanding of the resulting program by distributing closely related information throughout the code. To understand a piece of code, trails of method invocations had to be followed through many classes, moving both up and down the inheritance hierarchy. Soloway *et al*. [88] first observed this in the context of program comprehension, describing a 'delocalised plan' *as "where the code for one conceptualised plan is distributed non-contiguously in a program*". Soloway continues, "*Such code is hard to understand. Since only fragments of the plan are seen at a time by a reader, the reader makes inferences based only on what is locally apparent – and these inferences are quite error prone*".

Three significant issues were identified requiring further research:

- Chunking – how to partition a system for inspection

- Reading Strategy – the order in which the code is read
- Delocalisation – how inspections address the frequent references that object-oriented code makes to parts of the system that are not part of the current inspection focus

To address the latter two points, a systematic abstraction-driven reading technique was developed. The systematic technique forced inspectors to follow the trail of delocalisation, building up their understanding of the code. As this is achieved, inspectors create abstract specifications of each method. These can then be referenced by current inspections, future inspections, etc. An evaluation of the systematic reading technique comparing it against the ad-hoc reading technique found that there was no significant difference between the number of defects found by ad-hoc subjects compared to systematic subjects. However, some interesting issues emerged.

Defects with delocalised characteristics still appeared difficult to find. Subjects using the systematic technique found all the defects, whereas those using the ad-hoc technique missed several delocalised defects. As a by-product the systematic technique produced abstractions that can be reused at a later date for re-inspection. By generating the abstractions subjects found that they obtained a greater understanding of the code. Subjects commented favourably on the systematic techniques more structured process and ordering of the code when compared to the ad-hoc strategy, but found that the process of generating the abstractions required a lot of time. The systematic technique was found to help the weaker subjects, improving their defect detection ability, but was also found to inhibit the natural abilities of the stronger subjects. A potential weakness of the systematic technique was found to be its reliance on the static view of object-oriented code. The dynamic nature of object-oriented systems may hinder the effectiveness of such a static reading approach.

Three reading techniques were developed and compared to investigate these issues – a checklist (a traditional inspection approach), a systematic reading technique (evolved from the first evaluation), and a technique based upon use-cases (reads the code from a dynamic model viewpoint).

An evaluation of the three reading techniques found a significant difference (at the 10% level) in the number of defects detected between the reading techniques. The delocalised defects that were seeded in the experiment were more evenly distributed within the results for all the techniques.

The checklist technique was found to have the best overall performance, although subjects using the systematic technique were more effective at finding delocalised defects. Subjects noted that the checklist technique was easy and straightforward to use, however, several subjects suggested that the checklist did not deal with object messaging well enough.

Those who used the systematic technique stated that it encouraged a greater level of understanding. Subjects with different ability levels using the checklist performed reasonably well. The systematic technique was again found to help the defect detection ability of weaker subjects, but still seemed to constrain the ability of stronger subjects.

Weaker use-case subjects appeared to struggle (possibly due to the complexity of the technique). In general subjects found this technique very demanding. This may be a result of using students rather than subjects with more industrial experience. Some subjects suggested that one way to improve the use-case technique would be to introduce some form of checklist.

Roughly half of all the subjects using each reading technique suggested that the failure of the technique to find defects was not the fault of the particular technique itself, but due to their relative inexperience.

6.2 Lessons for the Inspection of Object-Oriented Code

This thesis has presented a large amount of information regarding the inspection of object-oriented code. Based upon this work, a series of recommendations can be made concerning object-oriented code inspection, as well as some general comments concerning reading techniques for inspection.

6.2.1 The Problem of Delocalisation

Effective reading techniques for object-oriented code inspection must address the issue of delocalised information. There is a substantial amount of evidence from the literature and the work presented in this thesis to support this view. Many of the features introduced by object-orientation, e.g. inheritance, polymorphism, dynamic binding, the use of small methods, all promote the distribution of information. Trying to understand one method becomes very difficult when so many other sources of information have to be investigated. Defects that involve ‘delocalised’ characteristics are the source of many of the ‘hard to find’ defects. For these defects, the amount of information that has to be read for the defect to be completely understood can become overwhelming and distracting. The reading techniques developed for this thesis attempted to address the problem of delocalisation, each taking a different approach.

The checklist is an established technique with a very simple procedure to follow – apply the questions in the checklist to the document under inspection. To address the issue of delocalisation, the questions used in the checklist were not general questions, but were derived from the historical defect information collected from the two previous experiments. This focused the questions in the checklist on areas of object-oriented code that were more likely to be associated with defects of a delocalised nature. The checklist was found to perform reasonably well for all defect types, suggesting that if checklists are tailored to the particular environment using historical defect data, and integrate questions that target object-oriented features, then they can be an effective aid to object-oriented code inspectors.

The systematic technique attempted to reduce the problem of delocalisation through the application of a reading order and the creation of abstractions. The reading order attempted to minimise interdependencies when reading the code. Creating abstractions forced inspectors to follow the trails of delocalisation and build up a sufficient understanding of the code. Subjects commented that, by using the systematic technique, they obtained a better understanding of the code. The systematic technique provided encouraging results concerning the detection of delocalised defects. The generated abstractions also provide a further way to reduce the problem of delocalisation. Once created, they can be reused in future inspections, localising the information required by inspectors and reducing the amount of code that has to be examined.

The use-case reading technique attempted to address delocalisation through the use of use-cases and sequence diagrams. Using these, the inspector is forced to consider the context in which an object is used. The technique also attempted to verify that the decisions and state changes made within each inspected method were correct and consistent. The results for the use-case technique were weaker than the other two techniques, possibly due to its increased complexity. More capable subjects may have been able to use the sequence diagrams and generated scenarios more effectively. However, the use-case technique does provide an independent source of comparison for the code with software requirements (in the form of use-cases, scenarios and sequence diagrams), which may help highlight defects of omission, which may themselves be delocalised in nature.

6.2.2 Reading Technique Overhead

Several recent publications have advocated the importance of inspectors understanding what they are inspecting [50], [79]. This has led to some recent reading techniques making inspectors carry out some form of task, i.e. creating use-cases, test cases, or class specifications. Comments from subjects participating in the three experiments presented in this thesis show that they prefer some form of guidance or structure when carrying out their inspections, and that this may help their understanding of the code. Although structure and guidance in reading techniques are useful in helping inspectors understand the artifacts under inspection, care must be taken not to overburden the inspector either with additional material or tasks to be performed.

The use-case technique was the most complicated out of the three investigated in the third experiment and was found to have the poorest defect detection performance. Subjects had to prepare a series of scenarios from the use-case, use each of the scenarios in turn to guide them through a sequence diagram, inspect methods in the code under review as they are found in the sequence diagram, and keep a note of system state information. Subjects found that the use-case technique was too slow and time consuming, that there was too much jumping between different inspection documents, and the generation of state information could become annoying and unwieldy. Subjects also found the sequence diagrams problematic, since their size restricted the amount visible at any one time on a monitor screen. All of this may explain the relatively low defect detection results that were found. It may be that, due to this complexity, the use-case technique is not one that can be used by novices and requires a more experienced software engineer.

The systematic reading technique was not as heavy on tasks or extra material as the use-case technique, but required subjects to follow the trails of delocalisation, build up a sufficient understanding of what the code was doing, and to write abstractions for each method. These abstractions could then be re-used in other inspections to help reduce the problem of delocalisation. Subjects commented both positively and negatively about the structure and strategy enforced by the systematic technique. The technique was found to help weaker subjects, improving their defect detection performance, but the structure imposed by the technique was found to inhibit the capabilities of the stronger subjects. Although the systematic technique has a higher overhead than the checklist, it has the added benefit of producing a set of abstractions that can be reused in later inspections and can help reduce the problem of delocalisation.

In comparison to the systematic and use-case technique, the checklist technique did not ask subjects to generate any extra material. Instead, it guided subjects to potential problem areas in the code via its questions. Subjects found this technique relatively quick and easy to use (possibly due to its low overhead). However, the checklist does not encourage the development of a deep understanding of the code.

Reading techniques require a balance, one that allows inspectors to concentrate on understanding the code and perhaps produce some useful documentation for later use, but without a large, distracting overhead.

6.2.3 Chunking

One of the three issues highlighted by the first experiment was that of chunking – how to partition a system for inspection. Due to the large number of dependencies within object-oriented code and the restrictions on the amount of code that can be looked at during inspection, it is difficult to isolate a reasonably sized section of code. To concentrate the subsequent experiments on the areas of reading strategy and delocalisation, the issues surrounding chunking were not investigated further. In the later experiments, for the most part, an arbitrary chunking solution was selected based upon inspecting classes as a complete unit. This was done for experimental reasons, to allow a fair comparison of the defect detection results between different reading techniques. This was achieved by making sure that each technique would roughly cover the same amount of code, or at least cover the same areas of the code where defects were present.

The use-case technique was the only technique that directly addressed the chunking issue. From use-cases, inspectors generated a series of scenarios, which were then traced on the sequence diagram by following the message calls between methods (moving horizontally through the system). On encountering a method for a class under inspection, the inspector switched their attention form the sequence diagram to the code. For a class under inspection, it was possible that not all the methods would be read, only those that were used by any one scenario.

The systematic technique did not directly address the issues of chunking, but did provide an inspection ordering for methods and classes in a system which attempts to minimise their interdependencies (coupling) by inspecting those classes and methods with least dependencies first. It was also suggested that, when minimising interdependencies, a class should not be split over more than one inspection. Although this does not exactly define what classes to chunk together to inspect, this provides a rough ordering with which to inspect the classes within a system and help inspectors build up an understanding of the system, especially when used in conjunction with the abstractions created by the process.

The checklist technique, as with the systematic technique, did not directly address the chunking issue, but the checklist technique was partially modified to be applicable to classes. This modification was based on ordering the questions in the checklist and grouping them into three categories: class (dealing with inheritance and constructor issues), method (dealing with all issues surround class methods), class (dealing with method overriding).

The work carried out in this thesis has not fully explored the issues and difficulties concerning the selection of code for object-oriented inspection. It may be that the best way to address the chunking issue is to select arbitrary classes, and let the reading technique deal with the consequences. More research is required to determine how best to chunk code, minimise the number of dependencies involved, and consider its impact upon the problem of delocalisation.

6.3 Advice on Practical Object-Oriented Code Inspection

The work presented in this thesis has made an initial investigation into the issues facing the effective inspection of object-oriented code. The main indication is that for inspections to continue to be effective, they must take into account the effect of delocalised information and the difference between the static and dynamic representation of code.

Checklist, despite their criticisms in the literature, can be very effective at this task. They are a relatively straightforward to use and have very few overheads. If checklists can be tailored to the development environment using historical defect data and include questions that specifically target object-oriented characteristics then they have the potential to be an effective aid to object-oriented inspections. However, it should be noted that this limits the checklist to recognised defect characteristics, and reduces the chances of finding new or unexpected defects. The questions used within the checklist should also try to encourage a more detailed understanding of the code and, in particular, its relationship with the rest of the system. This would help avoid the more traditional ‘lightweight’ checklist questions that only superficially probe the code.

The systematic technique provided encouraging results concerning the detection of delocalised defects. The technique offers a potential long-term advantage through the creation of abstractions. However, it has a higher overhead than checklists and may fail to adequately deal with some localised defects. Although the generated abstractions require further evaluation to establish their most effective form and usefulness, the ordering of code for inspection and the use of stepwise abstraction to help with delocalisation are aspects of the technique that can be recommended.

Although the results for the use-case technique were weaker, it has several potential strengths. Inspectors read the code from a dynamic model viewpoint and the technique offers an independent source of comparison for the code with software requirements (in the form of use-cases, scenarios and sequence diagrams). The technique better focuses on inter-class relationships as well as state information and has the potential to deal with defects of omission. This was found to be the most demanding of all the reading techniques, and it may be that it is a technique that can only be used by those with more industrial experience. However, it should be remembered that due to the nature of the technique, some parts of a class may go unchecked because they do not participate in the use-case that is driving the current inspection. It may be necessary to compliment this reading technique with another to ensure complete coverage of a class.

Where practical, object-oriented inspections should be based on teams of inspectors using at least two different reading techniques. The checklist was found to have a strong overall performance, but the systematic technique was found to be more effective at finding delocalised defects. A problem with the checklist is that its performance can heavily rely on the relationship between the questions and the context it is used in, whereas other techniques have less reliance on context and may give a more consistent performance. The work in this thesis also suggests that there is a need to take into account the dynamic model viewpoint.

Using a combination of reading techniques is a view similar to the one advocated by the developers of the Perspective Based Reading (PBR) technique, where different perspectives are used to represent different stakeholders, e.g. tester or code analyst. Each of these perspectives is expected to highlight different types of defects. If a PBR approach was to be adopted, it is suggested that one of the perspectives should specifically focus on object-oriented issues.

A further important consideration is how the techniques would scale up to deal with large amounts of program code. General advice in the literature suggests that the amount of code to be looked at in any one inspection should be limited to around 200 lines of non-commented code and the time allocated for this be around two hours. These restrictions should not change, no matter the size or scale of the system.

The systematic technique partitions a system in such a way as to minimise interdependencies, ideally not splitting a class over more than one inspection. Classes are ordered so that those with least interdependencies are inspected first. As inspections progress more and more abstractions are generated – ideally saving the inspector the effort of chasing delocalisation (by only reading the abstractions). A problem with the systematic technique may be that in larger systems, initially following the trails of delocalisation may be quite time consuming. This problem will however be reduced as more of the system is inspected and more abstractions are generated.

The checklist in this thesis was designed to be applied to a complete class, but can be applied to parts of classes as well. The key to the success of the checklist is the need for a well-maintained set of historical defect data being kept. The questions of the checklist are the main focus, and as such, scaling is not as much of an issue.

The use-case technique provides a dynamic, horizontal view of a system. The technique may encounter problems when scaling up to larger systems due to its use of sequence diagrams. The sequence diagrams may become large and unwieldy due to the increase in system size, and the time taken to traverse through them may become prohibitive. This does not prevent the technique being applied to systems of modest size. This issue may be alleviated by further tool development.

It should be noted that the reading techniques presented in this thesis do not attempt to address the issue of deciding which parts of the system should be prioritised for inspection. The reading techniques merely show ways in which the code can be read once selected.

Combining reading techniques, such as those highlighted in this thesis, offers a good degree of robustness and the potential to deal with many different defect types - the recurring defects, defects that require deeper insights, and defects associated with the features of object-orientation that distribute functionality throughout a system. However, this constrains the inspection process. A minimum number of inspectors are now required for complete coverage of the code.

6.4 Future Work

From the work carried out in this thesis there are several issues that require further investigation and experimentation.

For each of the three experiments, the subjects used were students participating in a third year computer science software engineering course. Using students can make results harder to generalise to a larger population, whereas using industrial subjects can greatly increase the validity of experimentation (due to their experience). One of the next steps in evaluating the usefulness of the reading techniques presented in this thesis is to use them in an industrial environment. A cost benefit analysis should be carried out to evaluate the effectiveness of the reading techniques, taking into consideration the amount of effort that is required, both to prepare the material necessary for the inspection and the amount of effort required by inspectors to use the techniques to inspect the code. This also presents a further opportunity to investigate the types of defect discovered by each technique.

As mentioned in the previous section, out of the three main issues identified as important to the successful inspection of object-oriented code, the issues of chunking – the selection of code for inspection – has not been fully investigated. The complete systems used in the experiments were relatively small (a few thousand lines of code). An interesting question for further study is how well the reading techniques would cope with a more realistic scenario where inspectors are reviewing 200 line 'chunks' from significantly sized object-oriented systems where delocalised references could lead deep into the rest of the system. More research is required to evaluate the impact chunking has on the inspection of object-oriented code, and to determine how best to chunk code, minimise the number of dependencies involved, and consider its impact upon the problem of delocalisation.

The systematic technique has shown encouraging results regarding the detection of delocalised defects. There are aspects of the technique that require further investigation:

- The systematic technique was found to hinder the natural abilities of stronger subjects, and at the same time helped the weaker subjects improve their defect detection. The reasons for the poor performance by stronger subjects are currently unclear, although it may be due to the systematic nature of the technique. Further work is required to investigate how this may impact on its use by industrialists.
- It is currently unknown what level of formality is required in the abstractions created during the inspection. Currently, natural language abstractions are generated by the technique. Further experimentation is required to decide if these are sufficient, or whether the precision and conciseness of semi-formal or formal specifications would provide greater benefits in terms of removing ambiguity.
- Further work is required to evaluate whether the abstractions are useful in reducing the problem of delocalisation and the amount of code that has to be examined.
- Tool support may make the creation of abstract specifications more efficient and reduce the burden of having to deal with the variety of documents that have to be managed during an object-oriented code inspection.

The use-case technique showed some potential as a candidate reading strategy but was found to be the most demanding of the reading techniques. The lack of knowledge and experience of student subjects compared to industrialists may have affected this technique more than the other two. Aspects that require further attention include:

- The technique was found to be overly complex due to its many different aspects: reading use-cases, generating scenarios, following sequence diagrams, and recording state information (some options to simplify the technique are highlighted in the following replication section).
- Novice inspectors appeared to struggle with the technique. A controlled experiment could compare the usage of the technique by novice and experienced inspectors and investigate whether the use-case technique requires a more advanced user.
- A tool may help reduce the complexity of the technique bringing together all the different sources of information and linking them together in a hypertext fashion, removing the paper overload that exists. A tool could also help make the sequence diagrams that are often large and difficult to navigate easier to read.

6.4.1 Replication Guidance

As well as further refining the reading techniques, it is important to explore the validity of the results that are presented in this thesis. This can be achieved via experimental replication (since this will require controlled experimentation, this will most likely be in a university environment) and then comparing the results with those contained in this thesis. To help with this all experimental materials, including all lecture and practice material have been collected together for each experiment and made available via the web for download at:

http://www.cis.strath.ac.uk/research/efocs/reports.html

Although these experiments can be replicated as is, there are several aspects that should be improved and modified.

In the third experiment, investigating three techniques at the one time reduced the number of subjects that were available. By not making the final experiment a 3x3 factorial design and having experimental subjects use just one technique the amount of data available to interpret was reduced. This was necessary due to time constraints. All of this helped reduce the reliability of the experimental results. It is suggested that any future experiments should only involve a maximum of two reading techniques and use a 2x2 factorial design (as was done for the second experiment in Chapter 4), especially if subject numbers are limited. This would allow for a detailed examination of two reading techniques with a reasonable amount of data to interpret. It should be noted that this type of experiment can have problems with a learning effect that can occur by subjects using more that one reading technique.

The code and the defects used in this thesis were created specifically for the experiments. The defects seeded were based on types highlighted in the literature and an industrial survey. To improve the validity of any future experiments, it would be preferable to use code and defects from an industrial source.

A constraint on future replication concerns the amount of practice required for reading techniques. Subjects using the systematic and use-case reading techniques in the second and third experiments commented that they would have liked more practice using the technique and more examples. In future replications, at least two practice sessions should be carried out per reading technique. As well as this, future experiments may benefit from having extra examples available for subjects.

In any future replication using the checklist technique, care must be taken with its content. The questions used in the checklist in this thesis were developed from a set of historical defect data. To be used in any other development environment, the questions should be based on historical defects from that environment. The notion of ordering the questions to help inspectors build up an understanding should be kept, as well as continuing to have questions specifically aimed at delocalised and object-oriented defect characteristics.

The use-case technique was found to be very demanding on the experimental subjects. An important area that requires further investigation is whether, due to its complexity, the use-case technique should only be used by more experienced software engineers. It may be that novices just require more practice in the technique, or that through simplification, the technique may become more manageable. One way to simplify may be to group all the methods together for a particular sequence diagram (to help reduce jumping around), and to reduce the amount of writing. Another way may be to assume that the scenarios for the system were developed as part of the requirement documents (along with the use-cases). This would reduce the time spent by inspectors reading the use-cases and generating the scenarios. However, this may also reduce the inspectors understanding of the system states and would assume that any scenarios previously developed as part of the requirements documents were correct.

One issue concerning the generation of abstractions in the systematic technique is what should be done where writing a method specification where defects have already been found. Including the defect may complicate the description that has to be written, and until the defect is removed, would remain in the abstraction (a problem if the abstraction was re-used in a later inspection). On the other hand, if the specification was not written, other defects still within the method may be missed. It is recommended that for any future replication, even if a defect is found, subjects should continue to create the abstraction, but should highlight the parts of it that are affected by the discovered defect(s). If an abstraction is read as part of another inspection before the defect is corrected, the inspector should assume that the method works correctly, but if any defects are subsequently highlighted, should detail any assumptions made based on abstractions used.

Future replication should also concentrate on several factors highlighted in the previous section concerning the systematic technique, i.e. validity and content of abstractions and the poor performance by strong subjects.

6.5 Conclusions

This thesis has shown that the way in which the object-oriented paradigm distributes related functionality can have a serious impact on code inspection and, to address this problem, it has developed and empirically evaluated three reading techniques.

Using a combination of reading techniques offers the potential to deal with many different defect types - the recurring defects, defects that require deeper insights, and defects associated with the features of object-orientation that distribute functionality throughout a system.

Bibliography

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, Software Inspections: An Effective Verification Process, *IEEE Software*, 6(3), pp. 31-36, 1989.

[2] J. T. Baldwin, An Abbreviated C++ Code Inspection Checklist, John T. Baldwin, University of Illinois, Department of Computer Science, October 1992 (available at http://www2.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html).

[3] V. R. Basili and H. D. Mills, Understanding and Documenting Programs, *IEEE Transactions on Software Engineering*, 8(3), pp. 270-283, 1982.

[4] V. R. Basili and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, *IEEE Transactions on Software Engineering*, 10(6), pp. 728-738, 1984.

[5] V. R. Basili, R. Selby, and D. Hutchens, Experimentation in software engineering, *IEEE Transactions on Software Engineering*, 12(7), pp. 733-743, 1986.

[6] V. R. Basili and H. D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Transactions on Software Engineering*, 14(6), pp. 758-773, 1988.

[7] V. R. Basili and S. Green, Software Process Evolution of the SEL, *IEEE Software*, pp.58-66, July 1994.

[8] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz, The Empirical Investigation of Perspective-Based Reading, *Empirical Software Engineering*, 2(1), pp. 133-164, 1996.

[9] V. R. Basili, Evolving and Packaging Reading Technologies, *Journal of Systems and Software*, 38(1), pp. 3-12, 1997.

[10] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F., Shull, S. Sørumgård, and M. Zelkowitz, *Lab Package for the Empirical Investigation of Perspective-Based Reading*, 1998. Available at http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html

[11] W. A. Belson, *The Design and Understanding of Survey Questions*, Aldershot & Gower, 1981.

[12] D. R. Berdie, J. F. Anderson, and M. A. Niebuhr, *Questionnaires: Design and Use*, 2nd Edition, The Scarecrow Press, 1986.

[13] R. V. Binder, Testing Object-Oriented Software: a Survey, *Software Testing, Verification and Validation*, Vol. 6, pp. 125-252, 1996.

[14] G. Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publishing Company, Inc., 1994.

[15] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[16] C 5.0, www.rulequest.com

[17] X. Chen, W. Tsai, and H. Huang, Omega - An Integrated Environment for C++ Program Maintenance, *International Conference on Software Maintenance*, pp. 114-123, 1996.

[18] B. Cheng and R. Jeffrey, Comparing Inspection Strategies for Software Requirements Specifications, in *Proceedings of the 1996 Australian Software Engineering Conference*, pp. 203-211, 1996.

[19] Y. Chernak, A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement, *IEEE Transactions on Software Engineering*, 22(12), pp. 866-874, 1996.

[20] R. T. Crocker, and A. von Mayrhauser, Maintenance Support Needs for Object-Oriented Software, in *Proceedings of COMPSAC'93*, pp. 63-69, 1993.

[21] J. Daly, *Replication and a Multi-Method Approach to Empirical Software Engineering Research*, PhD thesis, Department of Computer and Information Science, University of Strathclyde, Glasgow, UK, 1996.

[22] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software, *Empirical Software Engineering*, 1(2), pages 109-132, 1996.

[23] H. Deitel, and P. Deitel, *C How to Program*, second ed. Prentice Hall, 1994.

[24] M. Denscombe, *The Good Research Guide*, Open University Press, 1998.

[25] I. Duncan, D. Robson, and M. Munro, *Defect Detection in Code*, Testing Research Group, Computer Science, University of Durham, 1996.

[26] A. Dunsmore, M. Roper, and M. Wood, The role of comprehension in software inspection, *Journal of Systems and Software*, 52, pp. 121-129, 2000.

[27] A. Dunsmore, M. Roper, and M. Wood, Object-Oriented Inspection in the Face of Delocalisation, appeared in *Proceedings of the 22nd International Conference on Software Engineering 2000*, pp. 467-476, June 2000.

[28] A. Dunsmore, M. Roper, and M. Wood, Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented

Inspection, appeared in *Proceedings of the 24th International Conference on Software Engineering 2002*, pp. 47-57, May 2002.

[29] B. Edwards, *Statistics for Business Students*, First Edition, Collins, 1972.

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

[31] M. E. Fagan, Advances in Software Inspections, IEEE Transactions in Software Engineering, 12(7), pp. 744-751, 1986.

[32] R. G. Fichman and C. F. Kemerer, Object Technology and Reuse: Lessons from Early Adopters, *IEEE Computer*, 30(10), pp. 47-59, 1997.

[33] D. G. Firesmith, Testing Object-Oriented Software, published *in Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA, '93)*, 1993.

[34] W. Foddy, *Constructing Questions For Interviews and Questionnaires: Theory & Practice in Social Research*, Cambridge University Press, 1993.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley Publishing Company, 1994.

[36] T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley, 1993.

[37] R. B. Grady and T. Van Slack, Key Lessons In Achieving Widespread Inspection Use, *IEEE Software*, 11(4), pp. 46-57, July/August 1994.

[38] J. H. Hayes, Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach, *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS '94)*, 1994.

[39] M. Höst, B. Regnell, and C. Wohlin, Using Students As Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment, *Empirical Software Engineering*, 5, pp. 201-214, 2000.

[40] W. H. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[41] C. Jones, Gaps in the object-oriented paradigm, *IEEE Computer*, 27(6), June 1994.

[42] P. Jüttner, S. Kolb, and P. Zimmerer, Integrating and Testing of Object-Oriented Software, in *Proceedings of EuroSTAR'94*, 13/1-13/14, 1994.

[43] B. Kernighan and D. Ritchie, *Programming in C*, Hanser Verlag, 1990.

[44] E. H. Khan, M. Al-A'ali, and M. R. Girgis, Object-Oriented Programming for Structured Procedural Programmers, *IEEE Computer*, 28(10), pp. 48-57, October 1995.

[45] J. C. Knight and E. A. Myers, An Improved Inspection Technique, *Communications of the ACM*, 36(11), pp. 51-61, 1993.

[46] D. Kung, J. Gao, and P. Hsia, Developing an Object-Oriented Software Testing Environment, *Communications of the ACM*, 38(10), pp. 75-87, 1995.

[47] O. Laitenberger and J-M. DeBaud, Perspective-Based Reading of Code Documents at Robert Bosch GmbH, Special Issue on *Information and Software Technology*, vol. 39, pp. 781-791, 1997.

[48] O. Laitenberger and C. Atkinson, Generalising Perspective-based Inspection to handle Object-Oriented Development Artifacts, in Proceedings of the *21st International Conference on Software Engineering 1999*, pp.494-503, 1999.

[49] O. Laitenberger, C. Atkinson, M. Schlich, and K. El Emam, An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents, *The Journal of Systems and Software*, 53(2), pp. 183-204, 2000.

[50] O. Laitenberger and J-M. DeBaud, An Encompassing Life-Cycle Centric Survey of Software Inspection, *Journal of Systems and Software*, 50(1), pp. 5-31, 2000.

[51] O. Laitenberger, K. El-Emam, and T. G. Harbich, An Internally Replicated Quasi-Experiment Comparison of Checklist and Perspective-Based Reading of Code Documents, *IEEE Transactions on Software Engineering*, 27(5), pp. 387-421, 2001.

[52] O. Laitenberger and K. Kohler, The Systematic Adaptation of Perspective-Based Inspections to Software Development Projects, in proceedings of the *1st Workshop on Inspection in Software Engineering*, published by Software Quality Research Lab, McMaster University, pp. 105-114, July 2001.

[53] L. P. W. Land, C. Sauer, and R. Jeffery, Validating the Defect Detection Performance Advantage of Group Designs for Software Reviews: Report of a Laboratory Experiment Using Program Code, In *6th European Software Engineering Conference*, pp. 294-309, 1997.

[54] D. B. Lange and Y. Nakamura, Object-Oriented Program Tracing and Visualisation, *IEEE Computer*, 30(5), pp. 63-70, 1997.

[55] M. Lejter, S. Meyers, and S. P. Reiss, Support for Maintaining Object-Oriented Programs, *IEEE Transactions on Software Engineering*, 18(12), pp. 1045-1052, 1992.

[56] S. Letovsky and E. Soloway, Delocalised Plans and Program Comprehension, *IEEE Software*, 3(3), pp. 41-49, May 1986.

[57] K. J. Lieberherr and I. Holland, Assuring Good Style for Object-Oriented Programs, *IEEE Software*, 6(5), pp. 38-48, 1989.

[58] B. Lientz and E. Swanson, *Software Maintenance Management*, First Edition, Addison-Wesley, 1980.

[59] R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

[60] C. M. Lott and H. D. Rombach, Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques, *Empirical Software Engineering: An International Journal*, 1(3), pp. 241-277, 1996.

[61] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood, Applying Inspection to Object-Oriented Software, *Software Testing, Verification and Reliability*, Vol. 6, pp. 61-82, 1996.

[62] F. Macdonald and J. Miller, A Comparison of Tool-Based and Paper-Based Software Inspection, *Empirical Software Engineering*, 3, pp. 233-253, 1998.

[63] J. Miller and F. Macdonald, An empirical incremental approach to tool evaluation and improvement, *The Journal of Systems and Software*, 51, pp. 19-35, 2000.

[64] J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks, Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research, *Information and Software Technology*, 39, pp. 285-295, 1997.

[65] G. C. Murphy, P. Townsend, and P. S. Wong, Experiences with Cluster and Class Testing, *Communications of the ACM*, 37(9), pp. 39-47, 1994.

[66] National Aeronautics and Space Administration, Software Formal Inspection Guidebook, Technical Report NASA-GB-A302, National Aeronautics and Space Administration, 1993, http://satc.gsfc.nasa.gov/fi/fipage.html

[67] J. Nielsen and J. Richards, Experience of Learning and Using Smalltalk, *IEEE Software*, 6(3), pp. 73-77, May/June 1989.

[68] A. Oppenheim, *Questionnaire design, interviewing, and attitude measurement*, Pinter Publishers, new edition, 1992.

[69] D. L. Parnas and D. M. Weiss, Active Design Reviews: Principles and Practice, proceedings of *8th International Conference on Software Engineering*, pp. 132-136, 1985.

[70] A. A. Porter, L. G. Votta, and V. R. Basili, Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment, *IEEE Transactions on Software Engineering*, 21(6), pp. 563-575, 1995.

[71] A. A. Porter, H. P. Siy, and L. G. Votta, A Review of Software Inspections, *Advances in Computers*, 42, pp. 39-76, 1996.

[72] A. A. Porter and P. M. Johnson, Assessing Software Review Meetings: Results of a Comparative Analysis of Two Experimental Studies, *IEEE Transactions on Software Engineering*, 23(3), pp. 129-144, 1997.

[73] A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta, An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development, *IEEE Transactions in Software Engineering*, 23(6), pp. 329-346, 1997.

[74] A. Porter and L. Votta, What Makes Inspections Work, *IEEE Software*, 14(6), pp. 99-102, 1997.

[75] M. Priestley, *Practical Object-Oriented Design with UML*, McGraw-Hill, 2000.

[76] J. A. Purchase and R. L. Winder, Debugging Tools for Object-Oriented Programming, *Journal of Object-Oriented Programming*, 4(3), pp. 10-27, June 1991.

[77] M. Putaala and I. Tervonen, Inspecting Postscript documents in an object-oriented environment, *5th European Conference on Software Quality*, 1997.

[78] B. Regnell, P. Runeson, and T. Thelin, Are the Perspectives Really Different? - Further Experimentation on Scenario-Based Reading on Requirements, *Empirical Software Engineering: An International Journal*, 5(4), pp. 331-356, 2000.

[79] S. Rifkin and L. Deimel, Applying Program Comprehension Techniques to Improve Software Inspections, *19th Annual NASA Software Engineering Laboratory Workshop*, Maryland, 1994.

[80] M. P. Robillard and G. C. Murphy, Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, appeared in *Proceedings of the 24th International Conference on Software Engineering 2002*, pp. 406-416, May 2002.

[81] M. Roper and A. Dunsmore, Problems, Pitfalls and Prospects for OO Code Reviews, EuroSTAR’99, 1999.

[82] G. W. Russell, Experience with Inspection in Ultralarge-Scale Developments, *IEEE Software*, 8(1), pp.25-31, 1991.

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

[84] R. W. Selby, V. R. Basili, and F. T. Baker, Cleanroom Software Development: An Empirical Evaluation, *IEEE Transactions on Software Engineering*, 13(9), pp. 1027-1037, 1987.

[85] F. Shull, I. Rus, and V. Basili, How Perspective-Based Reading Can Improve Requirements Inspections, *IEEE Computer*, 33(7), pp. 73-79, 2000.

[86] M. Sinclair, Subjective assessment, in J. Wilson and E. Corlett, editors, *Evaluation of Human Work: A practical ergonomics methodology*, pp. 58-88, Taylor and Francis, 1990.

[87] R. Van Solingen and E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.

[88] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, Designing Documentation to Compensate for Delocalised Plans, *Communications of the ACM*, 31(11), pp. 1259-1267, 1988.

[89] P. Stevens with R. Pooley, *Using UML – Software Engineering with Objects and Components*, Addison-Wesley, Updated Edition, 2000.

[90] S. H. Strauss and R. G. Ebenau, *Software Inspection Process*, McGraw Hill Systems Design and Implementation Series, 1993.

[91] I. Tervonen, Consistent Support for Software Designers and Inspectors, *Software Quality Journal*, 5, pp. 221-229, 1996.

[92] T. Thelin, H. Petersson, and C. Wohlin, Sample-Driven Inspection, in proceedings of the *1st Workshop on Inspection in Software Engineering*, published by Software Quality Research Lab, McMaster University, pp. 81-91, July 2001.

[93] G. H. Travassos, F. Shull, M. Fredericks, and V. R. Basili, Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality, *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[94] L. G. Votta, Does Every Inspection Need a Meeting?, *ACM Software Engineering Notes*, 18(5), pp. 107-114, 1993.

[95] J. Welkowitz, R. B. Ewen, and J. Cohen, *Introductory Statistics for the Behavioral Sciences*, Second Edition, Academic Press, 1976.

[96] E. F. Weller, Lessons from Three Years of Inspection Data, *IEEE Software*, 10(5), pp. 38-45, September 1993.

[97] N. Wilde and R. Huitt, Maintenance Support for Object-Oriented Programs, *IEEE Transactions on Software Engineering*, 18(12), pp. 1038-1044, 1992.

[98] N. Wilde, P. Matthews, and R. Huitt, Maintaining Object-Oriented Software, *IEEE Software*, 10(1), pp. 75-80, 1993.