Practical Code Inspection for Object-Oriented Systems

Alastair Dunsmore, Marc Roper, and Murray Wood

Abstract-- This paper describes a series of three empirical studies devoted to the development of a rigorous approach for effective inspections of object-oriented (OO) code. Since the time that inspections were developed they have been shown to be powerful defect detection strategies. However, little research has been done to investigate their application to OO systems, which have very different structural and execution models compared to procedural systems. This suggests that inspection techniques may not be currently being deployed to their best effect in the context of large-scale OO systems. The studies reveal three significant issues that need to be addressed - the identification of chunks of code to be inspected, the order in which the code is read, and the resolution of frequent non local references. The sequence of experiments builds up a complement of three techniques: one based on a checklist, one focussed on constructing abstract specifications, and the last centred on the route that a use-case takes through a system. It is demonstrated that the checklist is the most effective approach but that the other techniques also have strengths and so for the best results in a practical situation a combination of techniques is recommended.

Index Terms--Object-Oriented, Code Inspection, Code Review, Empirical Methods.

1 INTRODUCTION

SINCE their inception over twenty-five years ago, inspections have become established as an effective and efficient means of detecting defects. This has been determined by a number of controlled experiments and a wealth of industrial case studies, and from their beginnings as code-based techniques, inspections are now applied to a wide range of document types from requirements and designs through to test plans. Over the years both the application of the technique and its supporting materials have been refined and honed and there is active interest in continually developing the concept. In spite of their broad application, there is a significant lack of information indicating how inspections should be applied to object-oriented (OO) code. Inspections were developed when the procedural programming paradigm was dominant, but the last ten years have seen the OO paradigm growing in influence and use - particularly since the introduction of C++ and Java.

This lack of guidance on how to apply inspections to OO code is disturbing. Object-oriented and procedural languages are different (admittedly, some more different than others), not only in their syntax but 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 - to name but a few. 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 adapt to this paradigm may inhibit the effective application of inspections to large-scale OO systems. Not only does the inspection technique need to adapt to accommodate the OO paradigm, but the wider process needs to be modified as the criteria for choosing "chunks" of material to be inspected must also change.

This paper reports the results of a long-term empirical investigation into the development of a strategy for OO code inspection. The study is based around three controlled experiments that have served to build up a rigorous OO code inspection technique. The first experiment focussed on raising potential problems and issues with OO inspections and identified the characteristics of "hard to find" defects. From this experiment, three significant issues were identified that are arguably crucial in order to make OO inspections practical for large-scale systems. These were: chunking (the mechanisms whereby a piece of code is selected for inspection), reading strategy (the order in which the code is read), and delocalisation (how inspections address the frequent references that OO code makes to parts of the system that are not part of the current inspection focus). As a result of this, a systematic abstraction-driven inspection technique was developed and evaluated with the second experiment. The results from this in turn lead to the development and empirical evaluation of two further techniques - one based on a checklist and the other based on use-cases - along with a refinement of the first systematic strategy, and it is the work carried out for the most recent evaluation that constitutes the majority of this paper. The rest of this paper consists of brief summaries of the first two studies (full details can be found in [3, 4]) followed by an analysis of the most recent study.

2 EMPIRICAL STUDIES

All three studies took place in a University environment using 3rd year honours Computer Science students who had at least two years programming experience (primarily Java in the second and third studies, and a mixture of Java and C++ in the first study). The inspectors all had brief prior experience of requirements inspection but not code inspection. The focus of all three studies was individual code inspection (although the third study did involve a group element as well) and all involved inspecting Java code at rate comparable to published industrial rates (~100 lines of code per hour in a 1.5-2.0 hr

• All authors are with the Department of Computer Science, University of Strathclyde, Livingstone Tower, Glasgow G1 1XH, Scotland, UK.
E-mail: {apd, marc, murray}@cs.strath.ac.uk

	8A *	9A	1A	7B *	5A	1B	4B	8B	2B	5B	6B *	3A	4A *	7A *	9B *	3B	6A *	2A	10A *	10B *
Use of class library													x	x					x	x
Inheritance/ implementation	x														x					
Wrong message												x	x	x		x				
Diagram mismatch	x			x											x		x			
Wrong object											x									
Override				x													x			
Data flow Error			x					x	x		x	x				x				
Instance variable misuse					x		x	x												
Locality (m, c, s)	s	m	m	s	s	m	s	m	m	m	m	m	s	s	s	m	s	c	s	s
Domain Knowledge	x			x																
Method size (s, m, l)	-	m	s	s	m	l	m	s	l	m	m	l	m	m	-	m	-	l	m	m
Algorithm/computation		x				x				x				x				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
% Discovered	100	100	94	91	89	87	87	83	74	74	74	67	50	50	48	48	42	33	0	0

Notes:

Defects ordered from left, starting with easiest to find
Locality – method (m), class (c), system (s)
Method size – 0-4 (s), 5-10 (m), 11+ (l)

Table 1 – Defects (columns) described by their features

session). Lectures and training in the techniques being evaluated were provided.

A threat to the validity of the studies exists concerning subjects used (3rd year computer science students) as they may not be representative of the general software engineering population (as they lack experience and maturity). This aspect was limited due to available resources.

2.1 Study 1

This first study involved 47 subjects using an ad-hoc inspection technique and was intended as an initial information gathering exercise into the issues related to inspecting OO code. The aim of the first study was to investigate the characteristics of defects that inspectors found difficult to detect. The defects were a mixture of naturally occurring and seeded based on information gathered from the literature. For each defect a series of characteristic keywords was compiled reflecting key features associated with the defect. Table 1 shows the defects, ordered by their discovery rate, together with their keyword characteristics (column 1).

During analysis of this information it was discovered that many of defects had characteristics that required some aspect of understanding *outside the class under inspection* to fully comprehend the defect (those with asterisks under their number in the top row). We termed this characteristic 'delocalisation' after Soloway's description of delocalised plans in program comprehension [9] - "*... code for one conceptualised plan is distributed non-contiguously in a program*". Soloway suggests that such 'plans' are difficult to understand because only fragments are seen at a time and the reader has to make guesses based on what is locally apparent.

```
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 1 - Java code for isRegistered method

In Table 1 the characteristics judged to be associated with delocalisation appear in the top six rows plus locality 'class' or 'system'. Notice how these characteristics cluster to the right of the table – the harder to find defects.

On consideration it becomes apparent that this delocalisation is a fundamental characteristic OO code. Key features of OO code – inheritance, dynamic binding, polymorphism, small methods and class libraries – distribute closely related information throughout the code. This can mean that the information required to understand one line of code, a method, or even a class is not wholly contained within the code under inspection, but is spread through other methods, classes, systems or libraries. (It should be noted that this problem may exist to a lesser extent in modularised procedural code).

A review of related literature, particularly the maintenance literature, revealed that a number of authors had already identified this feature of OO code (but not in an inspection context). Furthermore, a small-scale survey of professionals who had inspected OO code provided supporting evidence for this finding, from a larger-scale, industrial perspective [2].

To illustrate the concept of delocalised information consider the `isRegistered` method in Figure 1. When reading the method, the inspector needs to be aware of the delocalisation that exists within it. 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`?

This situation is by no means unusual, as OO programming is based around such message passing and the use of other classes and class libraries.

This study suggested that for inspections to be practical and effective for large-scale OO systems, OO 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. It may be that partitioning is not restricted to units of compilation (e.g. classes), but may be carried out in an orthogonal manner, e.g. using slicing. 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 OO code be read, especially if systematically reading and understanding all the code and its dependencies is impractical? Is there a reading strategy that could help inspectors deal with delocalisation? Can checklists or Perspective-Based Reading Techniques (PBR) [1] be modified to address delocalisation or are new reading strategies required?

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

2.2 Study 2

The second study involved 64 subjects and followed up on the findings of the first study by exploring a code reading technique that was specifically developed to address the issues of delocalisation and reading strategy. The basic idea for the technique came from that of Stepwise Abstraction [8]. The following describes the basic approach for the technique:

- 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.
- Classes and methods are inspected using an abstraction driven reading strategy. This involves reverse engineering an abstract specification for each method. This abstract specification may then be used both to support comparisons with the class specification, and also to support inspections which make subsequent reference to a previously inspected method.
- During inspection any references to external classes must be traced and understood. This may involve reading other methods, documentation, or previously created abstractions. This understanding is necessary to correctly specify each method.
- 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.

To develop the abstract specification, a deep understanding of each method is required. All aspects of the method should be systematically read and understood. All links to other classes should be understood. Development of this deep understanding may help create ‘the big picture’ and reveal more of the hard to find defects.

The abstract specification for each method should identify any changes of state and outputs in terms of inputs and prior state. The specification 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
- complete (cover all aspects of method’s functionality including that derived from references to other classes).

This second experiment compared the defect detecting capability of the systematic abstraction-based approach with a basic ad-hoc approach. Care was taken to seed an equal mixture of delocalised and non-delocalised defects. The main finding was that there was no significant difference between the systematic technique and the ad-hoc technique in terms of the average number of defects discovered, although there was a small improvement using the systematic approach. Using data gathered on the process followed by inspectors in reading the code it seems that ad-hoc inspectors performed two or three passes of the code building up their understanding whereas the systematic inspectors performed only one, or at most two, slower passes through the code.

Further analysis did uncover some potential benefits of the systematic approach:

a) Some defects remained completely undetected by any inspector using the ad-hoc technique, but this was not the case for the systematic approach. Although no group component (collation of defects) was carried out, the fact that the systematic technique found all defects might suggest that the group component would be more successful.

b) The systematic approach produced abstractions for every method as a by-product of the approach. It is intended

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?
…		
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?

Table 2 - Part of the Checklist

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.

c) There was anecdotal evidence from the subjects' questionnaires that the task of creating abstract specifications encouraged a greater understanding of the code under inspection.

d) The systematic approach provides an ordering for the reading strategy to deal with the delocalised, distributed nature of OO 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 they may lose their train of thought.

Finally, in this second study, there was further evidence that the delocalised defects were more difficult to discover than the localised defects.

One potential weakness of the systematic strategy (or any sequential reading strategy) may be that it is based on a static view of the code. Specifically, the inspectors 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 OO code is quite different from the static view. As Gamma *et al.* [5] state "*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.*"

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 seem to address adequately the highly dynamic nature of OO software and was found to be more time consuming.

The main findings from these first two studies were that delocalisation and the difference between the static and the dynamic views seem very real problems for the practical application of software inspection to industrial-strength, OO code.

2.3 Study 3

Following on from the second study, the third focused on a selection of reading techniques for OO code - systematic, checklist, and a use-case based reading technique. What follows are brief descriptions of each of these reading techniques, followed by the results of an experiment to evaluate the techniques.

2.3.1 Systematic

The systematic, abstraction driven technique used in the second study was re-used with some alterations, based upon feedback and observations. Instructions given to the inspectors were made clearer and more specific. Inspectors were also given more experience of the technique (via more training and examples). The information that inspectors had to write on their abstraction sheets was reduced (helping to speed up the process).

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

Table 3 - Use-case for Cancel Booking

2.3.2 Checklist

Checklists are a straightforward and commonly used technique to help with individual code inspection. Checklists 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 and should be phrased in such a way that if the answer is **No**, then a potential defect has been discovered. The checklist should be based on historical data [6, 7] and should not be a general checklist obtained from elsewhere as they can lose their relevance.

The checklist was developed from the experience gained in the two previous studies. It takes into account the structure of OO code and is ordered in such a way that supports inspectors in building up a thorough understanding of the code. The questions in the checklist are grouped into three sections:

1. Class - this section is concerned with inheritance and constructor issues.
2. Method - the middle section 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.

Part of the final checklist can be seen in Table 2. The following describes the basic approach for the checklist technique:

- Interdependencies (coupling) within the code under inspection are analysed and those classes with least dependencies are inspected first.
- The checklist contains two components – one highlights possible **features** of the code to concentrate on, the other provides **questions** to help identify defects for that feature.

Inspectors were told to begin by first applying the questions in the first class section of the checklist. Once completed, inspectors should move on to applying all the questions in the method section of the checklist to all the methods in the class, including constructors. Finally, once all methods in the class have been inspected, the questions in the second class section at the bottom of the checklist should be applied. This process is repeated for each class under inspection.

2.3.3 Use-case

The use-case reading technique attempts to address the dynamic nature of OO 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. Is it a good citizen of the system? More precisely, with respect to the use cases in which the object participates, to verify that:

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

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. Defects are discovered by noticing missing/incorrect methods, erroneous state changes etc. The principle behind the technique is that it forces the inspector to consider the context in which an object is used. This is in contrast to both the systematic and checklist approaches that consider a class in a more general context. The technique is intended to be complementary to other reading approaches, as it is likely that some parts of a class will not be checked (because they are not involved in that particular use-case).

To apply the technique, inspectors take each use-case in turn and devise a series of brief scenarios based on the preconditions, success and failure conditions, and the exceptions found in the use-case. A sequence diagram is used to guide them through the interactions that scenarios have with

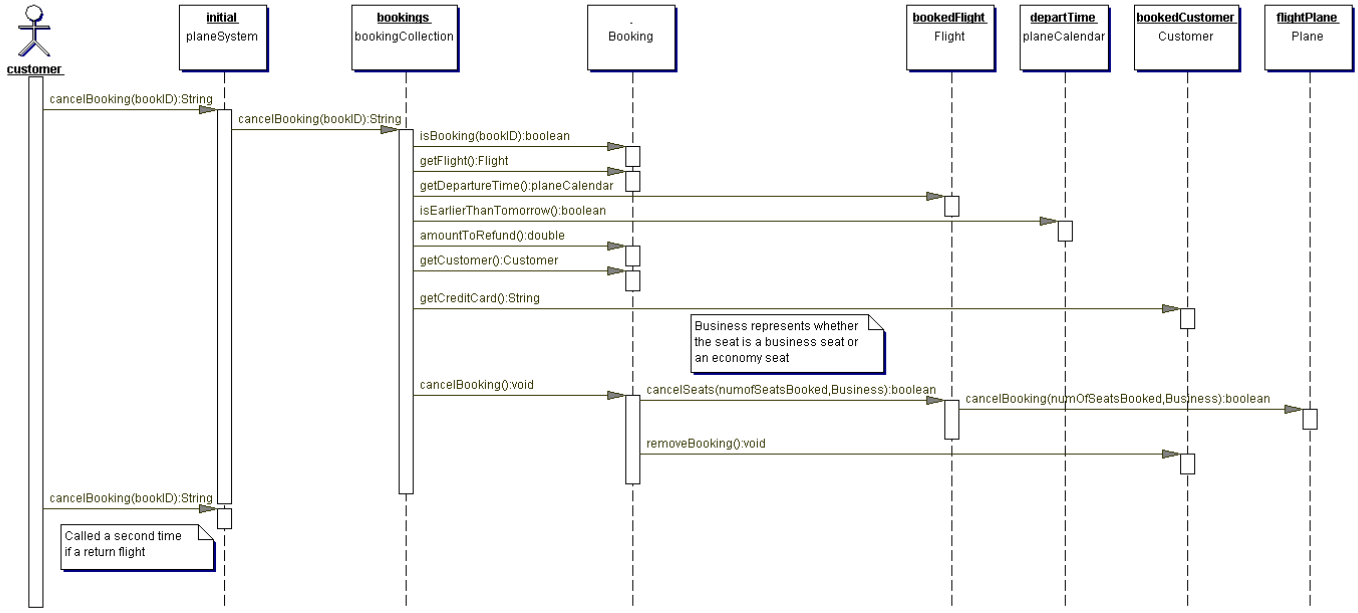

Figure 2 – Sequence diagram for Cancel Booking Use-case

the code under inspection. This requires that inspectors become familiar with the code under inspection, identify the state of the system that would cause this particular scenario to occur, identify the expected change of state and outputs from the class(es) under inspection as a result of the scenario, and to follow the scenario through the sequence diagram by tracing the message calls between objects.

If a class under inspection is encountered, inspectors verify that the expected methods are being called to support the scenario. When a method in the class under inspection is called any decisions and state changes that are made are verified to check that they are correct and consistent with respect to the scenario. While doing this, notes are made concerning any state changes and outputs. While inspecting a method, any method calls that are made are followed to verify that the correct ones are being called. If the method called is in the class under inspection, the call is followed and the method read, otherwise the inspector returns to following the sequence diagram.

Once through the scenario the final and predicated state/outputs are compared. If a difference exists between the two, then locate the source of the inconsistency and highlight as a candidate defect.

The following brief example highlights some of the concepts involved. Given the use-case shown in Table 3, the possible scenarios are:

		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 4 – Summary of experiment results for Study 3

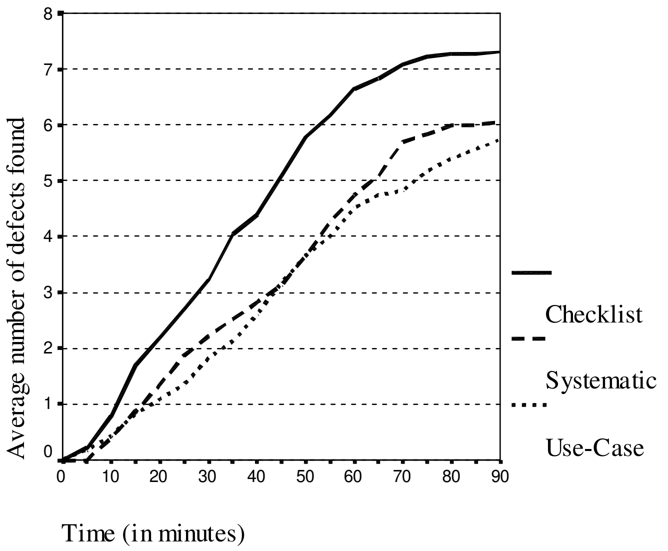

Figure 3 – Defect response rates

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

The class being inspected is the `PlaneCalender` class. Looking at the sequence diagram in Figure 2, the only method called in the class under inspection is `isEarlierThanTomorrow()`.

The anticipated state changes or outputs in relation to the developed scenarios are then noted:

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

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

Table 5 – Results of Kruskal-Wallis test

Finally, the code has to be inspected to verify whether the actual outcomes/state changes match those anticipated.

2.3.4 Results of Study 3

The third experiment compared the defect detection capability of the three reading techniques. Just over half of the defects seeded in the code (eight out of fourteen) were delocalised in nature. Table 4 presents a summary of the results from the third study.

Figure 3 shows the defect detection rates for each of the three reading techniques. Inspectors using the checklist technique appear to 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 inspectors appear to be fairly similar to each other, with systematic inspectors' performance levelling off towards the end of the 90 minutes. Use-case inspectors performance appears to be levelling off, but not to the same degree.

It would be expected that systematic inspectors would not have the quick detection rate of checklist inspectors since, as part of their technique, they are encouraged to fully understand the code and have to generate as a result of their understanding method abstractions, which can take time. Similarly, use-case inspectors have to generate scenarios from a use-case, and then using sequence diagrams, inspect the code (in a different order to other techniques).

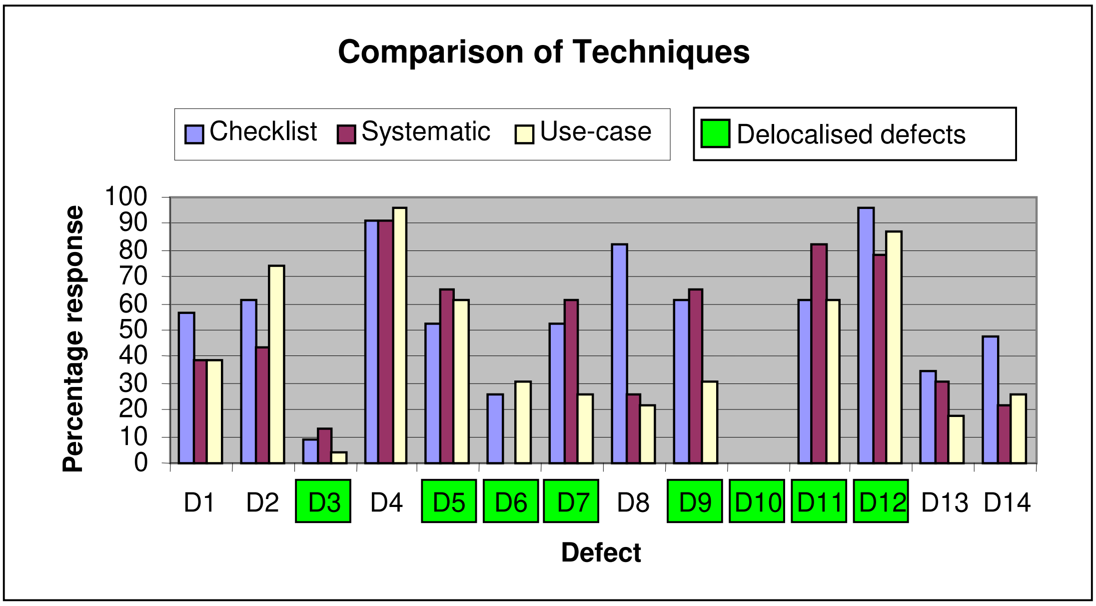

Figure 4 – All response rates for each defect

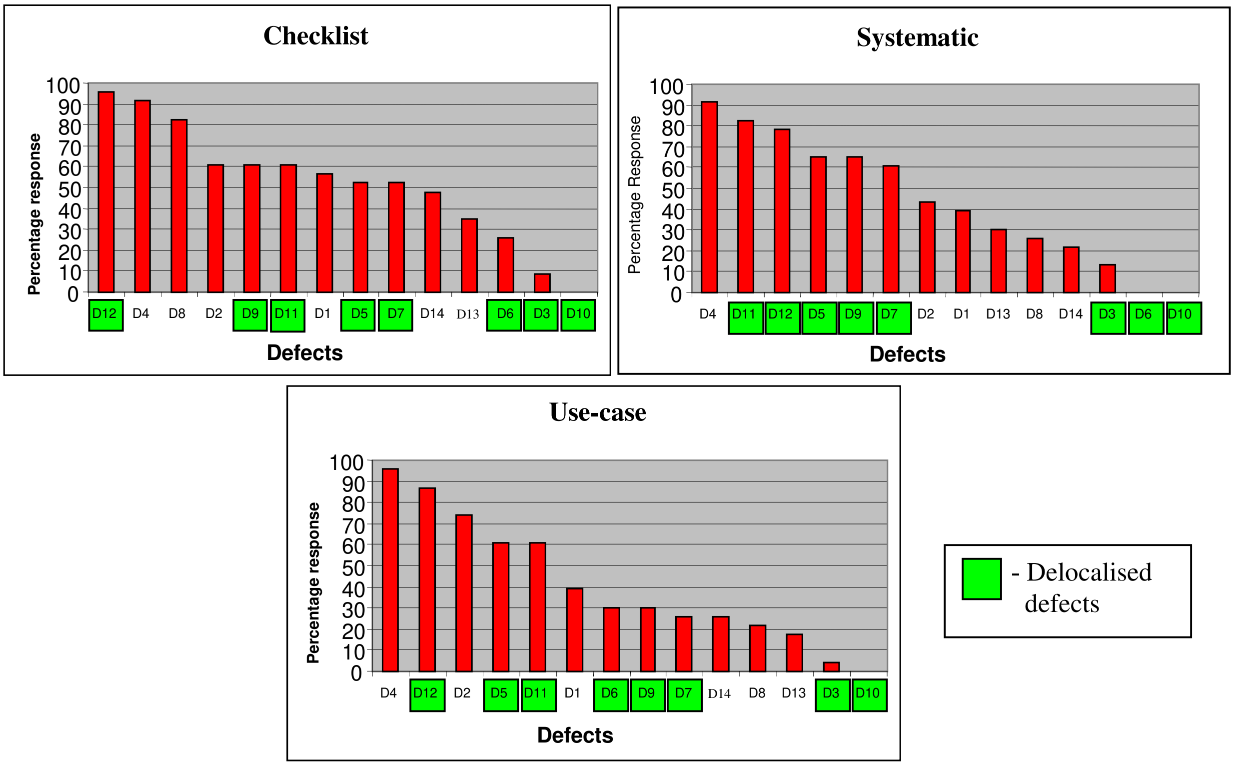

Figure 5 - Response rates split by technique

Due to the nature of the results it was not possible to apply parametric statistical tests, instead the Kruskal-Wallis test was used to determine whether the defect results for the three techniques were significantly different. The results generated by the software package SPSS are in Table 5. For 2 degrees of freedom, 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). There is a significant difference between the defect detection capability of the three reading techniques, but only at the 10% level of significance.

Further statistical tests showed that there was no significant difference between the three reading techniques in the number of false positives generated by inspectors.

Figure 4 shows the comparative effectiveness in defect detection for each of the three reading techniques. Those defect numbers along the bottom surrounded by a box are defects with delocalised characteristics. The checklist technique consistently appears to perform well. It should be noted that one defect (defect 10) was not found by any inspectors using any of the reading techniques. It was also noticed that defects involving some form of omission appear difficult to find (defects 6, 13 and 14). All the reading techniques have strong points, and there is not one dominant technique. This suggests a complimentary approach would work best, but more examination of the results is required.

Figure 5 shows three graphs, one for each technique. It represents the same information in Figure 4, but makes it easier to see the effect of each inspection technique on the delocalised defects. It is noticeable that the systematic technique appears to have detected the most delocalised defects with a response rate > 60%.

In conclusion, the initial results from the third study show that the delocalised defects are spread over the range of responses, suggesting that the techniques are having some effect, but that more analysis and experimentation is required to investigate more fully the strengths and weakness of the techniques.

3 CONCLUSIONS

This series of studies suggests that traditional reading techniques may not be appropriate in practice for effective inspection of large OO systems. There is evidence from these studies, and from the literature, that the key features of object-orientation can lead to problems of 'delocalisation' - having to understand related software that is not currently under inspection - and also require a deeper understanding of the dynamic view, as well as the static. In practice new methods of 'chunking', reading, and 'localising the delocalisation' seem to be required.

The studies explored the use of a systematic, abstraction-driven strategy, a specially created checklist and use-case driven strategy in an attempt to address some of these issues. There is some evidence to suggest that these techniques may be more effective in dealing with these problems:

- the delocalised defects were more evenly distributed within the range of easy and hard to find defects

- all the techniques seem to have some strengths in terms of finding different kinds of defects
- the abstractions developed as a side-effect of the systematic approach may be used in future inspections providing an alternative source of documentation and thereby contributing to the localisation of the delocalisation.

If time and resources are an issue and there is a well-developed repository of historical defect data then it maybe that tailored checklists provide the most efficient, practical approach. However, our belief is that, in practice, a combination of two or more techniques, perhaps incorporating further refinements of the systematic-abstraction and use-case based approaches, is most likely to provide the complementary views necessary to cope with both recurring defect types and the inevitable new, subtle defects which require deeper insights. This is part of ongoing research that still requires further analysis, experimentation and industrial confirmation.

REFERENCES

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

[2] A. Dunsmore, "Survey of Object-Oriented Defect Detection Approaches and Experiences in Industry", Technical Report – EFoCS-36-2000, Computer Science Department, Strathclyde University, August 2000.

[3] 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.

[4] A. Dunsmore, M. Roper, and M. Wood, "Systematic Object-Oriented Inspection – An Empirical Study", appeared in *Proceedings of the 23rd International Conference on Software Engineering 2001*, pp. 135-144, May 2001.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "*Design Patterns*", Addison-Wesley, 1994.

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

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

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

[9] 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.