An Empirical Investigation of a Systematic Object-Oriented Inspection Technique

A. Dunsmore

EFoCS-37-2000

Department of Computer Science
University of Strathclyde
Livingstone Tower
Glasgow G1 1XH

Email: apd@cs.strath.ac.uk

Abstract

Software inspection is a well-recognised defect detection technique, but recent research has highlighted that its level of performance on object-oriented code may be suffering due to the highly delocalised nature of the software. This paper presents the results of an empirical investigation, which compared the traditional ad-hoc inspection approach with a systematic, abstraction-driven inspection technique, designed to help with the problem of delocalisation.

The results showed no significant difference in relation to defect detection performance between the use of the ad-hoc and systematic inspection techniques. However, the systematic technique did seem to encourage a deeper understanding of the code being inspected and may also help discover different defects from the ad-hoc approach. Inspectors also seemed to appreciate the rigour that the systematic technique imposed.

This paper suggests that a systematic, abstraction-driven reading strategy offers some potential but that issues regarding the difference between the static and dynamic views of object-oriented code, as well supporting the efficient construction of abstractions needs to be addressed.

1 Introduction

Software inspection has over the years established itself as a credible technique for finding defects [6], however there is a growing collection of evidence to suggest that current techniques used during code inspection are not fully dealing with issues arising form the inspection of object-oriented code [4, 5, 9]. To date, the vast majority of reports appearing in the literature relate to inspections carried out with procedural languages, the predominant paradigm in use when inspections were first proposed in the 1970's.

The literature suggests 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 distribute closely related information throughout the code, leading to what Soloway has termed "delocalised plans" [12]. What this can result in is information required to understand one line of code, a method, or even a class not being completely contained within the code under inspection, but spread throughout other methods, classes, systems or libraries.

An experiment by Dunsmore *et al.* [4] further backs up the notion that delocalisation is a significant problem for the inspection of object-oriented code. It was found that 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. It was also found that defects with certain characteristics, e.g. wrong message, use of class library, inheritance, and overriding, all tended to be harder to find. Based on the results of this experiment and an industrial survey [5], 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

It was beyond the scope of this experiment to look at all three of these areas, instead the work presented here will focus on the latter two points. To address these, the authors developed a systematic, abstraction-driven inspection technique for object-oriented code, providing a strategy for reading code in a certain order (and hence attempt to minimise interdependencies) and a methodology that aims to increase an inspectors understanding of the code.

This paper presents the results of an empirical investigation comparing the defect detection levels of two inspection techniques, ad-hoc and the systematic technique, for object-oriented code. The following section describes the systematic technique for reading object-oriented code during an inspection. This is followed by section 3 detailing the design and section 4 presenting the results of the experiment. Section 5 presents an interpretation of the results and a series of improvements that might be made to the systematic technique, and the paper concludes with section 6.

2 Systematic Technique

There are a number of challenges when inspecting object-oriented code compared to traditional, procedural code. Object-oriented code has very frequent references to methods and classes that are not within the code being inspected, e.g. in libraries or other classes. What order should a collection of classes be inspected in? Is there an order that minimises interdependencies?

The two areas this technique is attempting to address are those of Reading Strategy and "localising the delocalisation". Reading strategy defines how we read through the code under inspection. Two forms of reading strategy are Systematic and As-needed. Soloway [12] described these 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)* [12].
As-needed Strategy: *Programmer 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* [12].

There are however problems with using these reading strategies for the inspection of object-oriented code. Systematically inspecting (and understanding) all code and its dependencies is likely to produce a cognitive overload; too much information to remember at the one time, whereas the as-needed approach to reading invites assumptions to be made and also precludes an inspector from gaining an overview of the code (no big picture). The challenge for localising the delocalisation is to find a way to provide the benefits (accuracy, complete information, etc.) associated with systematic reading, but with the efficiency of as-needed reading.

The philosophy behind the developed systematic technique is to have inspectors read the code in a systematic manner, and as they proceed through the code create abstract specification for each method as they read it. The aim of this is to make inspectors read the code systematically, following method calls and other outside information where appropriate, helping with the inspectors understanding, plus the beneficial side effect of generating a collection of abstractions which can be referenced by current inspections, future inspections, maintenance, etc. The following describes the basic approach for the technique:

- Interdependencies (coupling) 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 (see below for more).
- Classes and methods are inspected using abstraction driven reading strategy. This involves reverse engineering an abstract specification for each 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.

Notice that this 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 code.

As was mentioned in one of the bullet points above, methods within classes are analysed and those with least dependencies are inspected first. This was achieved by looking at each method and ranking the delocalised features of the methods. Those features included (ranked in order of growing delocalisation):

- Method call (to method (including class library methods) already looked at during the inspection)
- Parameter passed in / return type
- Casting
- Method call (in class under inspection)
- Method call (to class library method not looked at)
- Method call (outside current class, but in other classes under inspection)
- Method call (outside current class and not under inspection)

If a method had none of these points then it would be inspected relatively early on. These methods would be followed by methods featuring the first few points. As a method featured more of the later points, it would be more likely that the method would be inspected towards the end of the session. Methods inspected at the end would rely on methods not available, or methods inspected earlier on during the inspection session.

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 in question for each method should identify any changes of state (attributes/instance variables) and outputs (return values or messages) in terms of inputs and prior state (attributes/instance variables).

The specification should be:

- brief (as short as possible while capturing all aspects of the method)
- declarative (specification should 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).

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.

Specification for `UserCollection` class:

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.

Java Code for `isRegistered` method:

```
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;
}
```

When reading the method, the inspector needs to be aware of the delocalisation that exists within it. These are issues that require further understanding and should help when building up the abstractions, 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`?

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 example shows how an inspector can build up an understanding of the method following a stepwise reading approach. Linger, Mills and Witt [10] developed the stepwise abstraction technique of reading in the late 70's.

- `((Person)theUsers.elementAt(i)` gets the ith element from the vector `theUsers` and casts it to a `Person` instance. Can all users be casted 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()`. `String equals()` 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 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.

From all of this, a final abstract specification is written:

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

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.

3 The Experiment

The experiment compares the defect detection capability of the two inspection techniques, ad-hoc and systematic, for object-oriented code. The motivation for the experiment comes from the discovery that delocalised information may cause difficulty when inspecting object-oriented code and that current reading strategies do not adequately deal with this issue [4]. This section presents a brief description of the experimental design, followed by the major results of the experiment.

The ad-hoc reading technique with which a comparison will be made, offers no support to the inspector, instead the inspector has to rely on their own knowledge and experience, reading the code in their own preferred way.

To focus the aims of the experiment it was decided to use the Goal Question Metric (GQM) paradigm (originally developed by V. Basili and D. Weiss [1]) as described by Solingen and Berghout [11]. The GQM shifts the emphasis away from metrics to goals. The goals create focus for the important issues of an experiment. These goals are then specified in more detail by defining questions, which themselves suggest the appropriate metrics to be measured. The GQM can be seen as acting more as a guide, rather than a predefined set of rules and metrics.

The remainder of this section contains the information derived by the GQM technique, as well as full descriptions of the experimental design, preparation and procedures.

3.1 Goal/Question/Metric

3.1.1 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 stepwise 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:

1.1: H_0: There is no significant difference in the number of defects found by those subjects performing ad-hoc inspection or systematic inspection of OO code.

3.1.2 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:

1: -- : Exploration of results

3.2 Experimental Plan

Subjects

Subjects were participants in a 3rd year Honours Computer Science Software Engineering course run at Strathclyde University. 64 subjects were participating in the class. Subjects had previous experience with the programming languages of Java and C. The subjects had limited knowledge of Software Requirements Specification (SRS) document inspection, and no experience with code inspections.

Experimental Design

To investigate the two inspection 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. 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. Figure 1 shows the inspection order for the experiment. The experiment was based around a code inspection exercise. No group component was carried out, as the main focus of the experiment was the performance of the individual inspectors. The code inspections themselves were paper-based, no tool support was provided.

As part of an earlier exercise, subjects were split into 16 groups of 4, each group with approximately equal ability. The ability split was based on marks from previous classes. For this experiment, two groups of subjects were required. This was achieved by utilising the earlier 16-group split and randomly assigning each of them to the two new groupings. Figure 1 shows the inspection order for the experiment.

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

Figure 1. Inspection order

Material

Code

The language chosen for the code documents was Java. There are several reasons for this choice. The experimental design required that an object-oriented language be used. A further reason was that the subjects had a reasonable knowledge of the Java programming language (having used it for the preceeding 2 ½ years) and it would make the inspection unrealistic if the subjects were to inspect a code document in a programming language they have little or no experience with.

The code documents presented to subjects for inspection were approximately 200 lines in length. Fagan [6] suggested that a maximum of 125 non-commentary source statements per hour are read. Weller [13] from information gathered from over 400 inspections suggested no more than 200 lines of code per hour. Gilb and Graham [8] suggest at most 1 ½ pages (approximately 90 lines of code) per hour. Information gathered by Dunsmore [5] from an industrial survey found that on average more than 200 lines of code were being inspected per session. Although it is difficult to obtain a clear consensus, the literature generally agrees that inspecting too much code reduces the effectiveness of code inspections. For the purposes of this experiment (which will last 90 minutes), the code documents used will be of the order of 200 lines of code, bearing in mind that students are being used, and not professionals.

Prior to the experiment, subjects had been given a problem statement describing a hotel booking system. 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.

For the practice sessions of the experiment, subjects were presented with material used in an experiment that had been run the previous year [3]. 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). Subjects had not previously seen any code documents or specifications for these extensions.

Defects

Another important aspect of the material to be used was the defects that were to be present within the code documents. The defects were based on several sources; a previous experiment investigating object-oriented inspections by Dunsmore *et al*. [4], information collated from the literature [3], an industrial survey [5], and a selection of naturally occurring defects (i.e. appeared in code when written by one course lecturer). In total ten different defects were seeded into each of the two code documents. Since the experiment was interested in 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 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 and above each method in the code booklet, a series of boxes in which to enter the time that subjects began reading that method.

A questionnaire was prepared and given to subjects upon completion of the inspection. The aim of the questionnaire was to gather extra information on resources used and problems encountered by subjects during their inspection, as well as gauging their performance.

For systematic inspections, inspectors were given both a defect report form and code booklet (exactly as for ad-hoc inspections) and were also given method specification sheets. These contained boxes in which subjects were to write their abstract specifications for the inspected code. As before, once completed the inspection, subjects were given a questionnaire. As well as asking about resources used, problems encountered and gauging performance, this questionnaire also asked about subject's opinion on the systematic technique and what they thought were the advantages/disadvantages over ad-hoc inspection.

Data Analysis

The goals of the experiment feature both testable hypothesis and exploratory analysis. Where appropriate SPSS was used to test the hypothesis using independent sample t-tests and Levine's test for equality of variance. The remaining goals that were exploratory in nature were investigated through the analysis of the qualitative information gathered during the experiment.

3.3 Experimental Procedures

Based on the experimental design, 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 Activities

In weeks 1 and 3 an introductory lecture and training phase were carried out. Each lecture lasted approximately 50 minutes and introduced all of the relevant information and techniques. The next day, a training session lasting 1 ½ 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.

Conducting the Experiment

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). If a subject went beyond the time limit they were asked to stop.

Once subjects had finished the inspection task, they were supplied with a questionnaire. Subjects were given approximately 20 minutes to complete the questionnaire. Both the inspection task and the questionnaire were completed under exam conditions.

Threats to Validity

An empirical study can suffer from influences which may affect the experimental variables without the knowledge of the researcher (threats to internal validity). This possibility should be minimised as much as possible. These include:

- Selection effects, which can occur through variations in the natural performance of individual subjects. 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.
- There was no way to monitor the preparation of subjects prior to the experiment as they worked in groups. Some of the subjects within groups could have worked at different rates, taken on more responsibility (covering for less well able subjects or subjects who did not participate fully). This could lead to an imbalance of subjects knowledge and understanding of the system prohibiting them from performing during the inspection experiment and skewing some of the results.
- 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 as much as possible by retaining all paper material after each experiment. Subjects were also never informed before or after the experiments 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 4 weeks subjects were carrying out an inspection per week. It is possible that subjects found this repetitive and interest dropped of towards the end. To try and counteract this, course credit was awarded to subjects for completing the inspection exercise and the questionnaire.
- Learning effect. Due to possible learning effects, ad-hoc inspections had to be carried out for both groups of subjects before systematic inspection (necessitating the use of both sets of code each week – see previous).

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

- The subjects of the experiment (3rd year computer science students) may not be representative of the general software engineering population. This was limited due to available resources.
- The Java code may not be representative (in complexity or stylistically) of industrial software. In this case though the code inspected was part of a substantially large software system, diminishing some of the complexity issues.
- The defects seeded in the code may not be representative of the problems currently experienced in industry. As mentioned earlier, this has hopefully been overcome by basing defects on information from various sources (literature, industrial survey, previous investigation).
- The inspection process used during the experiment may not have been representative of industrial software practice. This experiment focused only on the defect detection phase (for the individual), used ad-hoc and systematic methods for inspecting the code documents, and did not involve any presentational overview by the author, group collation phase or rework phase.

4 Results and Analysis

64 subjects participated in the experiment. Due to reasons of absence from the practice run or absence from at least one of the proper inspection exercises, the results will be based on 53 subjects. Three other defects were discovered for the gym code document 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.

4.1 Defect Detection

Figure 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 an improvement in the mean number of defects found between the ad-hoc and systematic techniques, however the difference between the two is not statistically significant (based upon an independent sample t-test – see Figure 3). This further backed up by Levine's Test for Equality of Variance (see Figure 3). This means we have to accept the null hypotheses for 1.1, that there is no significant difference in the number of defects found by those subjects performing ad-hoc inspection or systematic inspection of OO code.

	Gym		Conference Room	
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 (mean)	3.44	3.86	3.04	3.44
No. of Defects (St. dev)	2.0632	2.4603	1.7947	1.4166
No. of Defects (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

Figure 2. Summary of defect detection results

	Significance (2-tailed)	F
Gym	.509	.514
Conference Room	.371	.099

Figure 3. T-Test and Levine's Test for Equality of Variance

Shown in Figure 4 and Figure 5 are the defect detection rates for both the hotel system extensions. In both figures, those using the systematic technique are slower at finding defects (probably due to extra time required to write specifications). In Figure 5 there is a large difference between detection rates initially. This could be due to the fact that this extension was written in 3 classes. Following the systematic technique meant reading through the classes in a certain order. The first class they would have read only had 1 defect and the second class only had 3 (out of a possible 10). Those who were inspecting the 3 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 Figure 4, where the rates are fairly close.

In both diagrams the systematic technique at some point obtains a better detection rate than the ad-hoc technique. By the end of the ad-hoc inspections, the detection rate appears to be dropping off, whereas the drop off for systematic inspections is not quite as severe (the drop off for systematic in Figure 4 is due to there being no data point between 95 and 100 for systematic; there was for ad-hoc).

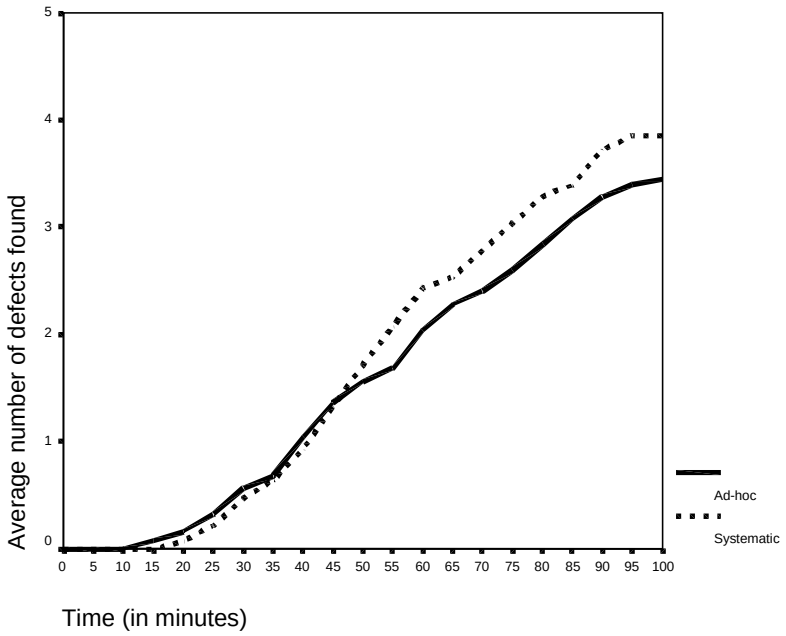


Figure 4. Gym defect detection rate

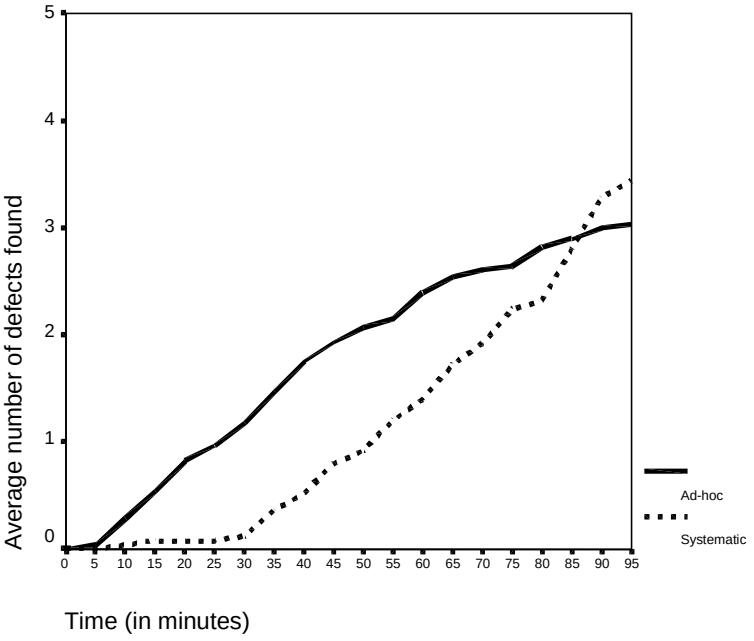


Figure 5. Conference Room defect detection rate

Shown in Figure 6 and Figure 7 are the defect detection rates for delocalised defects. There were 5 delocalised defects present in each of the code documents. In both cases, those using the systematic inspection technique are slower to find delocalised defects (due to time creating written specifications) but as time goes on, they eventually surpass the detection rate of those using the ad-hoc inspection technique.

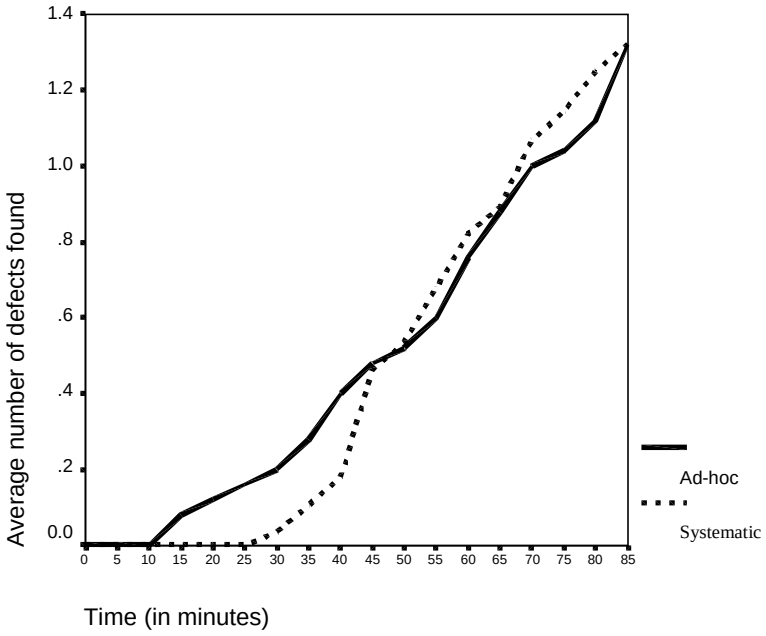


Figure 6. Gym defect detection rate for delocalised defects

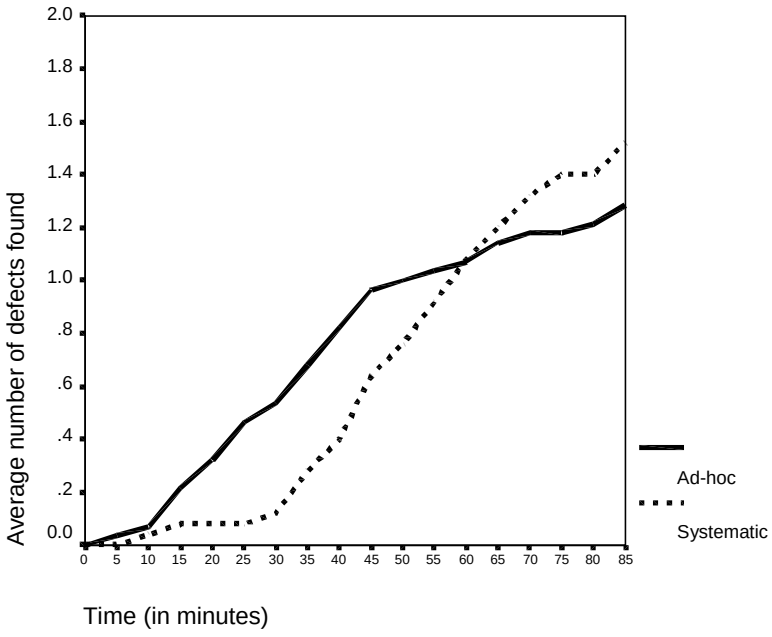


Figure 7. Conference Room defect detection rate for delocalised defects

Figure 8 and Figure 9 show the defect detection rates for the non-delocalised defects. There were 8 non-delocalised defects present in the gym code document and 5 in the conference code document. In both cases the defect detection rate is very low. This may be explained by the fact that very few of the defects, including the non-delocalised defects were trivial in nature, and that a thorough understanding of the code documents were required. For the gym code document, the defect detection rates are reasonably similar, but for conference the systematic technique takes much longer to get going. The reason for this is due to the class and method ordering. There were three classes to inspect and because subjects were using the systematic technique they had to read the classes in a specific order. The first

two classes to inspect only contained 4 defects, 3 of which were delocalised in nature. This would explain why it takes such a long time for inspectors to find non-delocalised defects, as the bulk of them were contained in the last class subjects had to inspect.

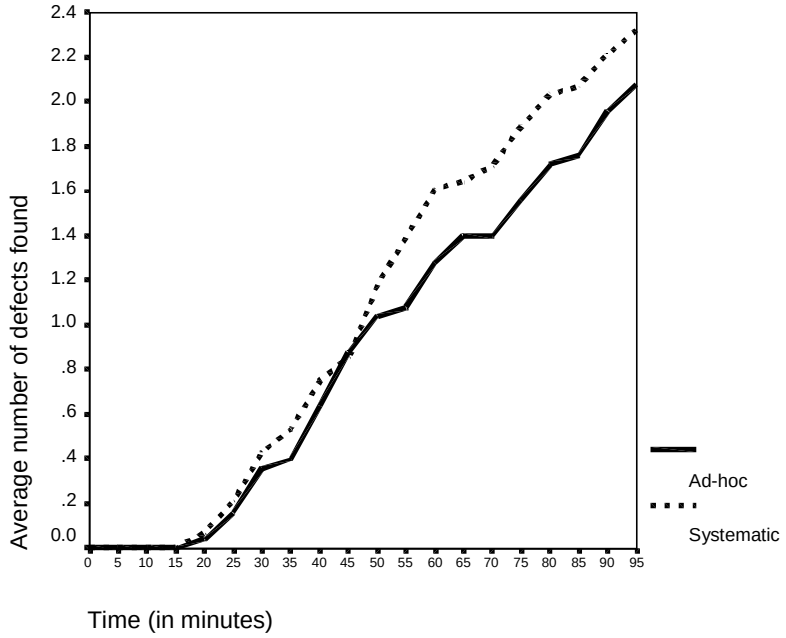


Figure 8. Gym defect detection rate for non-delocalised defects

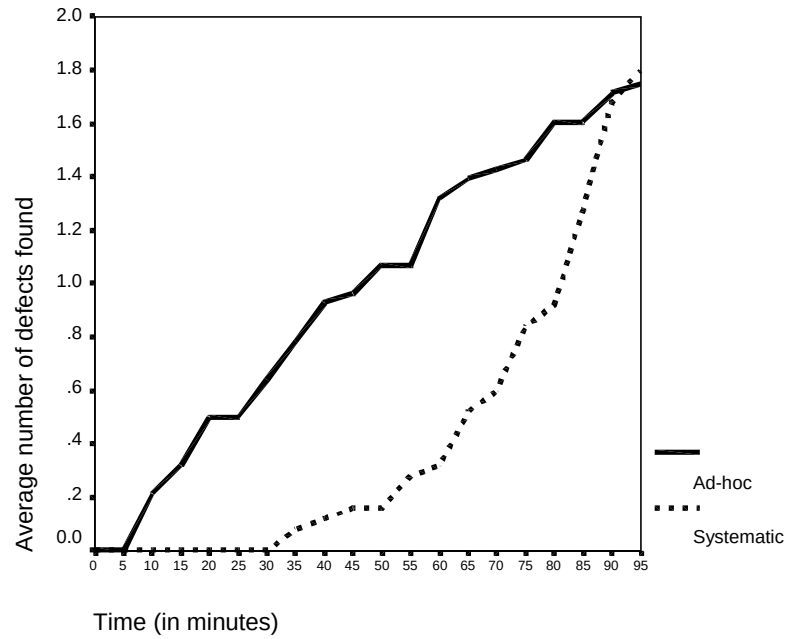


Figure 9. Conference Room defect detection rate for non-delocalised defects

4.2 Reading Strategy

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 (see Figure 10 for example). This information was then compiled and entered into a small purpose built tool to help visualise how subjects read through the code. An example of this can be seen in Figure 11. It shows in what order the classes and methods were read, as well as approximately how long was spent reading them. This example was taken from an ad-hoc inspection. This particular subject read through the code using a combination of two techniques, reading the code in the order presented and following method calls.

```
// [   :   ] [   :   ] [   :   ] [   :   ] [   :   ] [   :   ] [   :   ]
  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 10. Example of code and time boxes

Figure 12 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 were reading the code two to three times. In comparison, 30-40% of subjects when reading via systematic were reading the code only once, with the vast majority of the rest reading the code less than twice. Subjects were spending longer reading methods as they attempted to fully understand what they were doing and create their abstractions. Whereas with ad-hoc inspections, subjects would repeatedly browse through the code multiple times, sometimes spending very little time concentrating on each method.

As part of the questionnaires given to subjects after completing their ad-hoc inspection, they were asked to select from several descriptions the one that best described their reading strategy. These results are shown in Figure 13. 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 they believed needed re-inspection. It appears that in the later stages of the inspection, subjects reading order is not affected by code order or method calls. Most of the remaining subjects read the code by following method calls, and having read through the code this way at least once, again used this information to decided which methods to revisit later on. Only about 17% of all 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 (see section 2 for full details). Nearly all subjects stuck to reading the code in the suggested order. About half of those 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 (sometimes following a method call) before returning to the correct order. More time was spent by all subjects reading methods for the first time, than was the case with ad-hoc inspection. After having read the code at least once, it was still the case that about half the subjects revisited certain methods towards the end of the inspection, but it was to a far lesser extent than that for ad-hoc inspection. About 9% of the subjects failed to complete their inspection of the code (i.e. were not able to read all the methods in the code).

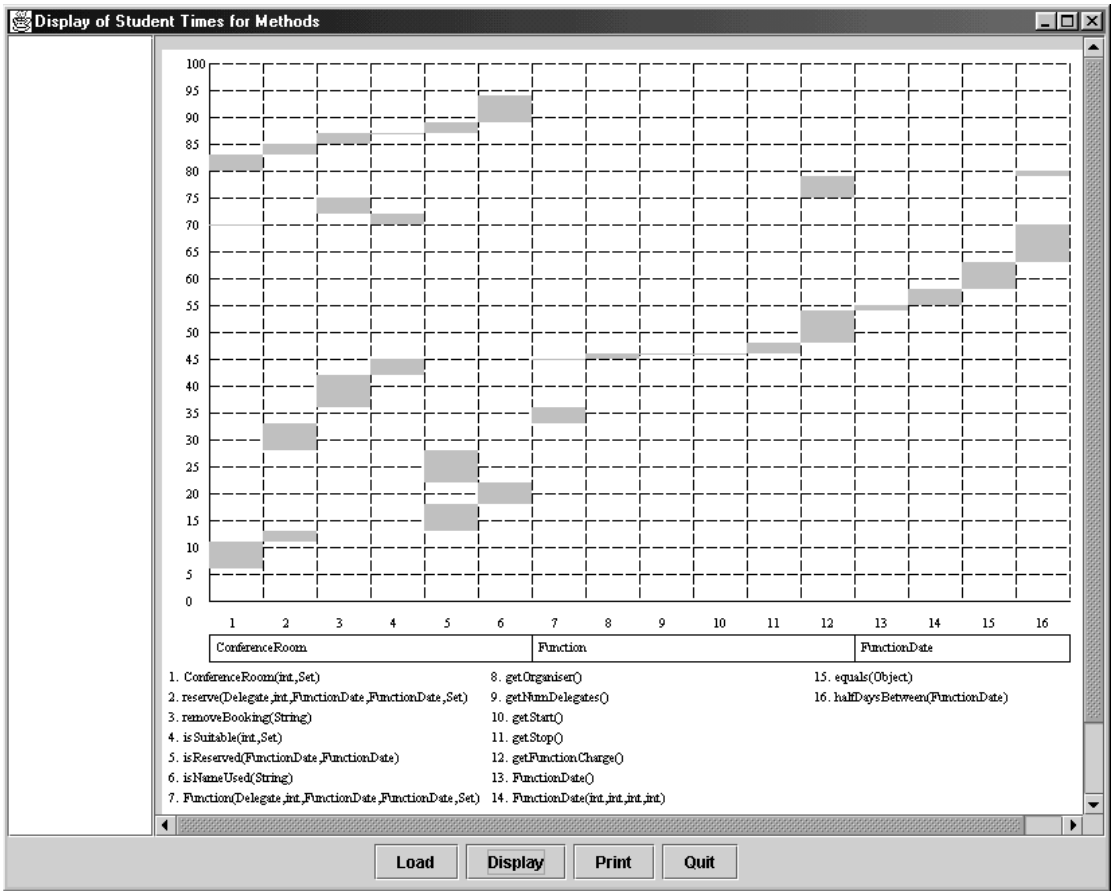


Figure 11. Screen shot of a subject's reading strategy

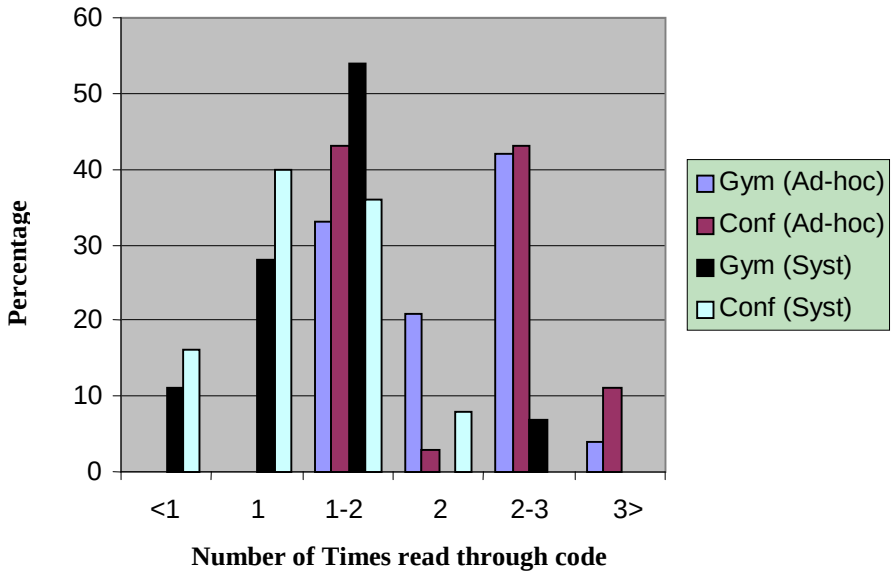


Figure 12. Number of times subjects read through the code

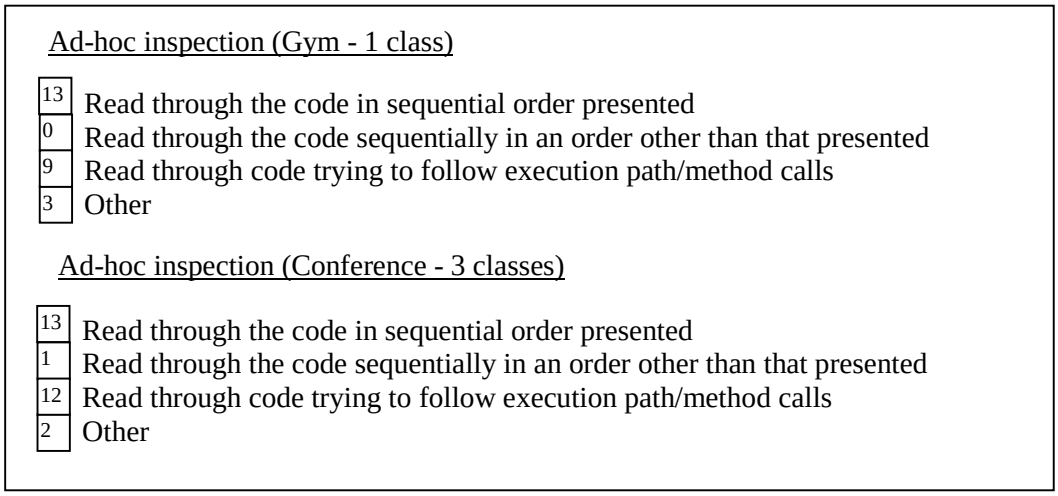


Figure 13. Ad-hoc reading strategy

	> ½ defects found on first run through code	> ½ defects found on subsequent runs through code	Same number of defects found on first and subsequent runs
Gym (ad-hoc)	54%	21%	25%
Gym (systematic)	75%	4%	21%
Conference (ad-hoc)	54%	32%	14%
Conference (systematic)	100%	0%	0%

Figure 14. When defects where found by subjects in relation to their reading strategy

Figure 14 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. For the two ad-hoc groups 21% and 32% of subjects found more than half of their defects on subsequent passes through the code, and 14% and 25% of subjects found the same number of defects in their first pass and subsequent passes through the code. There is a slightly higher percentage of subjects finding more than half of their defects in subsequent passes for the multiple class code document (conference – 32%) compared to the single class code document (gym – 21%).

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, and 21% of subjects reading the gym code found the same number of defects in both their first and subsequent passes through the code. This significant increase in finding more defects in the first pass through the code is not surprising considering that subjects were spending longer reading each method while creating their abstractions and were making fewer passes through the code (see Figure 12).

	Ad-Hoc	Systematic
Defect 1 (Calendar) – prepared HotelDate class / found defect	6 / 3	7 / 2
Defect 1 (Calendar) – read Calendar class library / found defect	7/ 2	17 / 8
Defect 7 (ListIterator) – read LinkedList class library / found defect	22 / 5	24 / 8
Defect 7 (ListIterator) – read ListIterator class library / found defect	19 / 6	20 / 6
Number of subjects who found defect 1	8	9
Number of defects who found defect 7	6	9

Figure 15. Gym class library usage

	Ad-Hoc	Systematic
Defect 2 (Calendar) – prepared HotelDate class / found defect	7 / 1	6 / 1
Defect 2 (Calendar) – read Calendar class library / found defect	17 / 2	19 / 0
Defect 4 (Calendar) – prepared HotelDate class / found defect	0 / 0	6 / 2
Defect 4 (Calendar) – read Calendar class library / found defect	17 / 0	19 / 2
Number of subjects who found defect 2	3	2
Number of defects who found defect 4	0	8

Figure 16. Conference class library usage

Several of the defects present in each of the code documents relate directly to information obtainable from the Java class libraries. Defects 1 and 7 are an example of this in the Gym code document (see Figure 15). Defect 1 involved the use of an incorrect constant in the `Calendar` class. In the coding exercise previous to this experiment, some subjects coded a class (`HotelDate`) that involved the use of the `Calendar` class and could possibly have come across the relevant information which would help to spot the defect. Figure 15 shows that half or less than half of the subjects who wrote the `HotelDate` class actually found this defect. Only 2 subjects out of 7 for ad-hoc inspectors (17 out of 8 for systematic inspectors) found this defect after having at some point read the `Calendar` class library documentation during their inspection. Also shown are the results for defect 7, which used a method from the `ListIterator` class incorrectly. Very few subjects would have had previous contact with this class (highlighting why most subjects bothered to look at it). Approximately 30% of subjects who read the `ListIterator` class also found defect 7.

Similar information is also shown in Figure 16 for the Conference code document. It is a similar sort of picture, but with most subjects reading the `Calendar` class library. In all cases very few subjects appear to be finding the defects, even after reading the class library.

From the information contained within Figure 15 and Figure 16 it appears that subjects were not finding the information they require from the online documentation. The Java online documentation although competent, can lack clarity, support for queries and useful examples. Subjects seemed either not able to navigate the documentation successfully or unable to comprehend the information presented to them.

Figure 17 and Figure 18 show the reading strategies used during the ad-hoc inspections by the best and worst performers (based on number of defects found). There is no outright reading strategy used by the subjects who performed well during the inspection, with some subjects following method calls, and the remainder reading the code in the order presented to them. All of the subjects who performed badly during the exercise read the code in the order presented.

What can also be seen in Figure 17, is that those who performed well during the ad-hoc inspection did not significantly improve their performance when carrying out the systematic inspection. 8 of the 12 subjects approximately obtained the same percentage of defects for both inspections, 4 of the subjects performed notably worse during the systematic inspection than in the ad-hoc inspection. This indicates that the systematic technique was possibly constraining the natural abilities of the better subjects, 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, Figure 18 shows for the subjects who performed the worst during the 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 subjects struggling.

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

Figure 17. Subjects who found most defects in their ad-hoc inspections

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

Figure 18. Subjects who found the least defects in their ad-hoc inspections

4.3 Questionnaire Results

A questionnaire was given to each subject after they had completed their inspection (for both ad-hoc and systematic).

When reading code during an ad-hoc inspection 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. Problems highlighted by subjects included it did not help with understanding, presented less structure to the inspector and subsequently there was more jumping around the code. Subjects using the ad-hoc inspection technique stated that it was less time consuming, less restrictive in its reading order (more freedom) and easier compared to the systematic technique. Suggestions on how to improve on inspection performance concerned better knowledge of language/class library, more information relating to code under inspection, the ability to compile the code, access to tools and aids, and more practice.

With the systematic technique subjects found that reading the code in the suggested order meant there was less jumping around, they gained an improved understanding, and that the process was more structured/focused. The down side found by following the ordering was that it could sometimes feel restrictive, with major methods being left to the end of the ordering, and one or two commented that it was slower and took longer to read through the code. When creating the abstractions, subjects suggested that it encouraged understanding and made you read each line of code, that subtle defects were easier to identify, and that instead of having to re-read methods you could read your previously written abstractions. Problems writing the abstractions were that it was time consuming, there was too much to write, and that in several cases it was difficult to write english specifications based on the code. When asked what could be done to improve systematic inspections, subjects main concerns were more time, more practice, and more information/examples at lecture.

4.4 Defects

	Gym		Conference Room	
Defect No.	Ad-Hoc	Systematic	Ad-Hoc	Systematic
	% Raw	% Raw	% Raw	% Raw
1	32 (8/25)	32 (9/28)	79 (22/28)	80 (20/25)
2	40 (10/25)	46 (13/28)	11 (3/28)	8 (2/25)
3	24 (6/25)	43 (12/28)	4 (1/28)	12 (3/25)
4	4 (1/25)	11 (3/28)	0 (0/28)	32 (8/25)
5	8 (2/25)	4 (1/28)	29 (8/28)	36 (9/25)
6	20 (5/25)	11 (3/28)	75 (21/28)	32 (8/25)
7	24 (6/25)	32 (9/28)	25 (7/28)	32 (8/25)
8	12 (3/25)	25 (7/28)	39 (11/28)	32 (8/25)
9	32 (8/25)	43 (12/28)	0 (0/28)	12 (3/25)
10	72 (18/25)	64 (18/28)	43 (12/28)	68 (17/25)
11	64 (16/25)	46 (13/28)	/	/
12	12 (3/25)	11 (3/28)	/	/
13	0 (0/25)	18 (5/28)	/	/

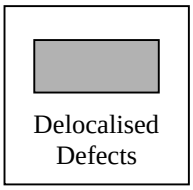


Figure 19. Raw defect detection results

Shown in Figure 19 are the raw defect detection results for the experiment. Defects deemed to have delocalised characteristics are highlighted in grey. This information is reformatted graphically in Figure 20. For the ad-hoc inspections, the three defects that are not found by any subjects all have delocalised features. 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 author, but were found by subjects during the inspection. One of those, defect 13 was only found by the systematic inspection technique.

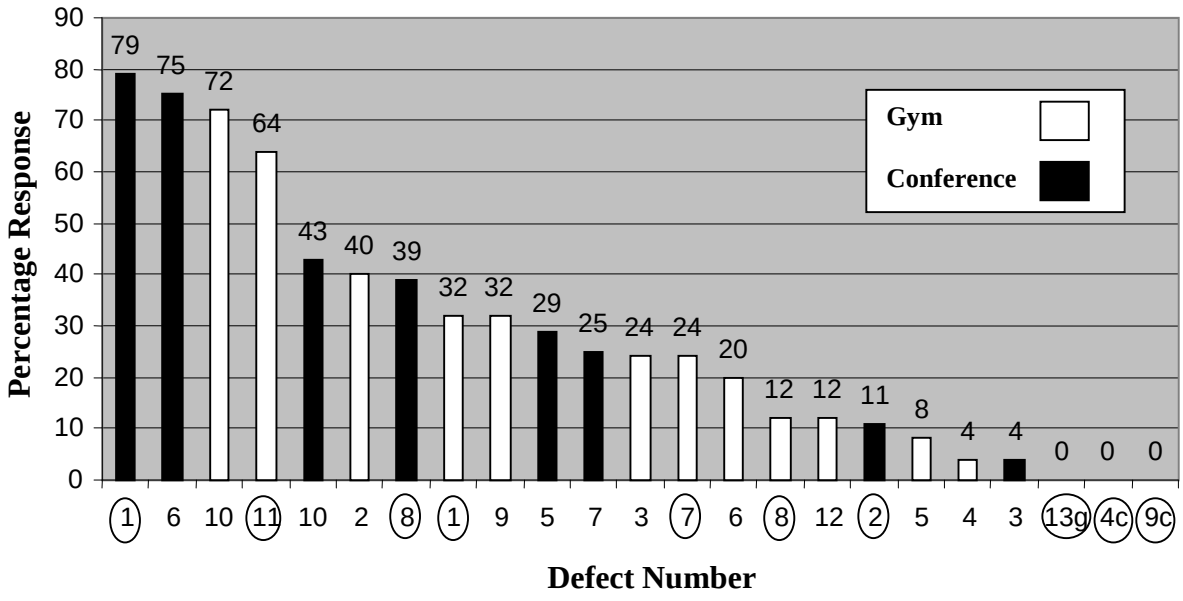
Two defects that stand out with high detection rates are defects 1 (conference code document) and 11 (gym code document). Defect 1 involved the call to the wrong method on an object. One possible reason for the high response rate may be due to the fact that the defect is highlighted by an inconsistency with the specification supplied to subjects. Defect 11 involved an erroneous line of code. The line could be considered reasonably obvious as it did not make any sense within the context of what the method was to do. Although defect 11 was found by 64% of ad-hoc inspectors, it was only found by 46% of systematic inspectors.

For the systematic inspections, there are no defects left unfound. The same two defects as highlighted for the ad-hoc inspections were again the defects with the two highest detection rate for systematic inspections. The detection of delocalised defects in some cases is improved, but is not statistically significant.

There were three defects contained within the gym code document that were not originally seeded by the author, but were highlighted by subjects during the inspection. One was non-delocalised in nature, the second was partially delocalised (within the class only), and the third was a delocalised defect. The delocalised defect was one of those not found by the ad-hoc inspectors, but found by systematic (defect 13).

Shown in Figure 21 is the frequency of detection for defects of 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.

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

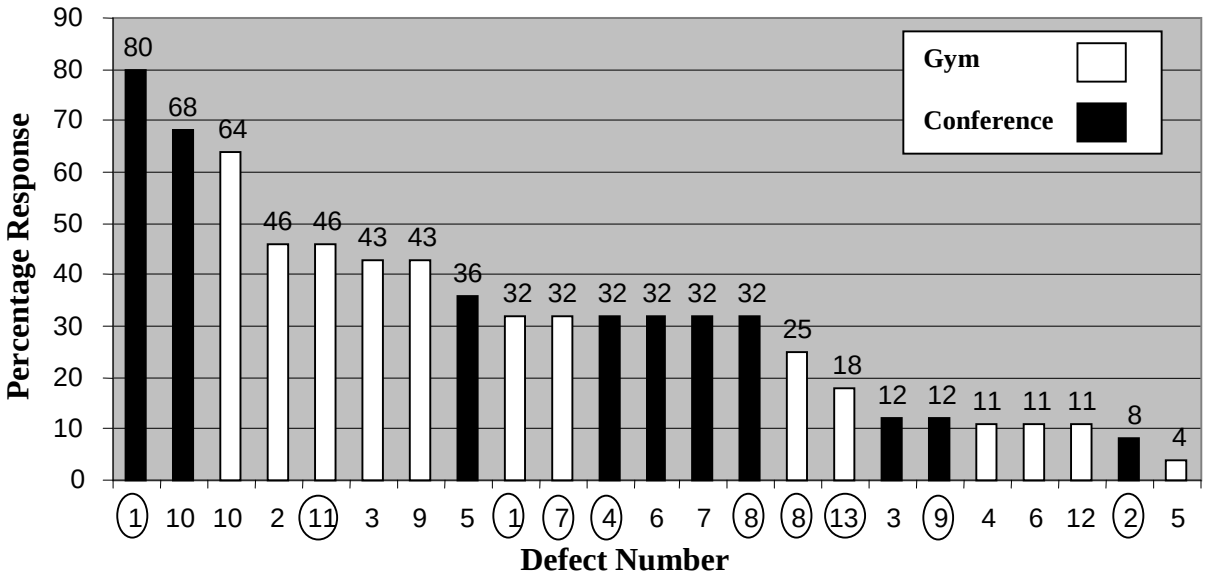
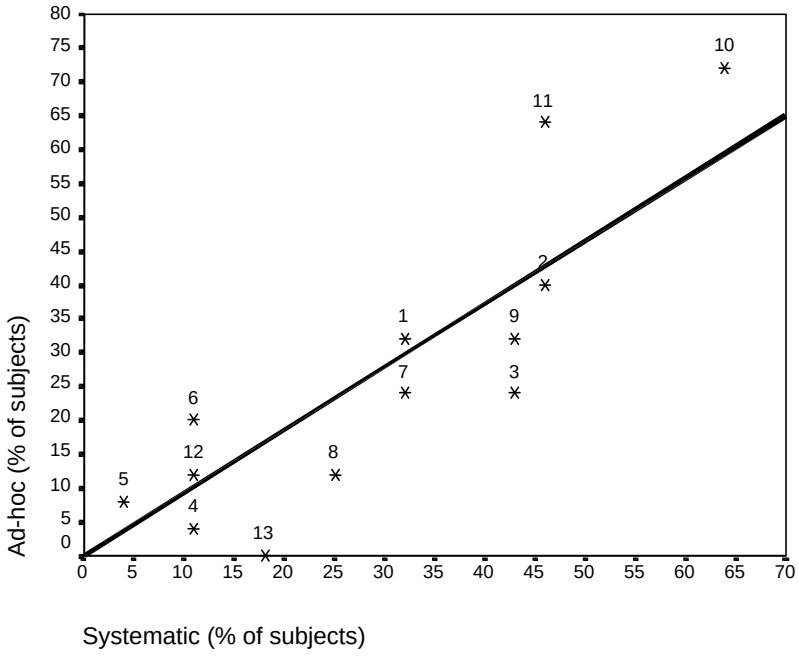


Figure 20. Percentage of subjects finding each defect for both code documents and defect detection techniques

Gym



Conference

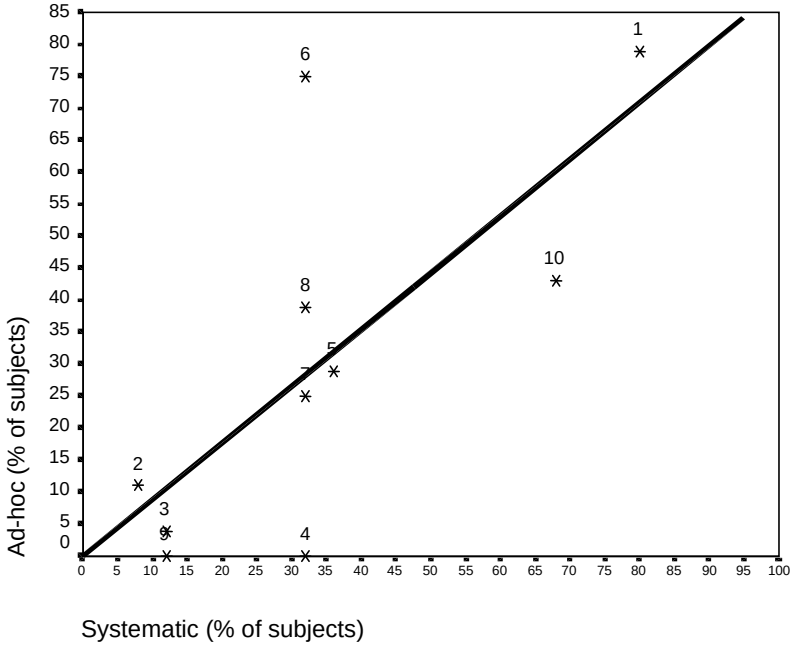


Figure 21. Frequency of detection

For the Gym code document two defects that stand out are 11 and 13. Defect 11, as already mentioned involved an erroneous line of code and was found by 64% of ad-hoc inspectors and by 46% of systematic inspectors. Defect 13 involved a check on the value returned by a call to another object. There was a problem with the check due to the value returned by the other object. 0% of ad-hoc inspectors and 18% of systematic inspectors found this defect. Both of these defects contained delocalised features.

For the Conference document two defects that stand out are 4 and 6. Defect 4 involved the call to the wrong method from a class library. 0% of ad-hoc inspectors and 32% of systematic inspectors found this defect. Defect 6 was a delocalised defect. Defect 6 involved the incorrect ordering of parameters in a call to a method. 75% of ad-hoc inspectors and 32% of systematic inspectors found this. The method being called belonged to another class under inspection. One possible reason for the unexpected low score for the systematic inspectors is that they may have felt they could not look back at previous code because of the enforced ordering. What is surprising is that they didn't think to look back at their abstraction sheets, which show the parameters expected for each method, and the order they are expected in.

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 misuse														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

Defects (Both code sets) – Ad-hoc Inspection – Key Descriptors

Gym

Conference

Defect No.	1	10	10	2	11	3	9	5	1	7	4	6	7	8	8	13	3	9	4	6	12	2	5
Locality (M,C,S)	S	M	M	C	S	C	C	C	S	S	S	S	M	S	S	S	M	S	C	M	C	S	M
Method (S, M, L)	L	M	L	L	L	L	M	M	L	L	M	M	M	M	L	L	M	L	L	L	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 misuse																	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	
%	80	68	64	46	46	43	43	36	32	32	32	32	32	32	25	18	12	12	11	11	11	8	4

Defects (Both code sets) – Systematic Inspection – Key Descriptors

Figure 22. Characteristics for all defects in defect detection order for both inspections

For each of the defects (23 in total) over both code documents, a list was drawn up of their characteristics. A description of each of the characteristics is shown in Figure 23. Figure 22 shows the characteristics for the defects in percentage response order for both the ad-hoc and systematic inspections. From the information presented in Figure 22, and the results of entering the information into C5.0 [2], a rule induction system (a rule induction system attempts to draw out a series of rules (or patterns) from information supplied to it), the following points were observed:

- Both method size and defect locality are fairly well mixed
- For both inspection methods, defects involving wrong object and instance variable misuse were difficult to find

- Defects involving data flow errors and class library access were mostly difficult to find, with one or two exceptions
- Defects involving a clash with the specification were for both inspection methods in the higher response end of the tables
- There were slight improvements from ad-hoc to systematic inspection for defects concerning wrong message and use of class library
- Defects involving omission were never found by more than 46% of the subjects.
- Both tables for ad-hoc and systematic are reasonably similar, and contain similar trends.

Defect Descriptors

Locality - area of code required to be looked at to spot 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 miss-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

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

Delocalised – defect contains characteristics which make it delocalised

Specification clash – defect highlighted by clash with specification

Figure 23. Description of defect characteristics

Where as Figure 17 and Figure 18 showed the performance of the best and worst subjects for ad-hoc inspections, Figure 24 shows the defect detection levels of the top and bottom subjects, based upon their marks from previous courses (the information used to split the subjects into their groups for the earlier section of this course). This shows that for the top 15 subjects, half achieved higher than average defect detection rates, the other half about average. For the bottom subjects, only one achieved higher than average defect detection rates, the remaining subjects were split between average and less than average performance.

Subject	Gym (ad-hoc)	Conference (syst.)	Gym (syst.)	Conference (ad-hoc)
1	6			5
2		3	6	
3	7			6
4		5	4	
5		3	6	
6	4			6
7		7	9	
8	3			4
9		6	7	
10	2			2
11		4	8	
12	2			4
13		2	3	
14	4			3
15		6	8	

Subjects ranked (highest first) from previous course work (top subjects)

Subject	Gym (ad-hoc)	Conference (syst.)	Gym (syst.)	Conference (ad-hoc)
1	0			2
2	2			2
3	3			2
4		3	5	
5	2			6
6	2			3
7	1			3
8		3	3	
9		0	2	
10		6	6	
11		0	2	
12		1	2	
13		2	2	

Subjects ranked (highest first) from previous course work (bottom subjects)

Figure 24. Defects discovered by top and bottom ranked subjects based upon previous course work

Defect 6 involved two parameters passed in the wrong order to a method. It is not surprising that defect 6 is found more by those following method calls, since this would bring the defect more to the attention of the inspector. This also highlights one of the reasons perhaps why subjects using the systematic technique were less successful (see Figure 20) at finding this defect; where they were reading the methods in the order given to them, and not following calls to other methods in other classes under inspection.

5 Interpretation

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 and Figure 5), although there is a small improvement for both code samples using the systematic approach. This means we cannot reject the null hypothesis, i.e. there is no significant difference in the number of defects found by those subjects performing ad-hoc inspection and 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 of the approach. 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.
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 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 6) when inspecting the multi-class code (the Conference code). 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 outputs 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 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 20) 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] and delocalised references were only to neighbouring classes or to the Java class library. 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 Gamma *et al*. [7] 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 adequately address the highly dynamic nature of object-oriented software, may be more time consuming and may restrict the natural ability of experienced or skillful 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 20 shows that the delocalised defects are, in the main, to the right (found by less than 39% of subjects only). 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%).

It seems that delocalisation and the difference between the static and the dynamic views of object-oriented code are very real problems for the application of software inspection.

6 Conclusions

This paper described the evaluation of a systematic, abstraction-driven inspection technique for object-oriented code. 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. The research 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 refinements of this systematic approach can provide a pragmatic reading strategy which 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. There is therefore a need to explore alternative strategies that are more in tune with the dynamic nature of object-oriented systems such as those that follow use-cases or dynamic execution slices.

References

1. Basili, V. R., Weiss, D. M., A methodology for collecting valid software engineering data, *IEEE Transactions on Software Engineering*, Vol. 10, No. 6, 1984.
2. C5.0, www.rulequest.com
3. Dunsmore, A. *The Role of Comprehension and Defects in OO Inspection*, Technical Report – EFoCS-34-99, August 1999.

4. Dunsmore, A., Roper, M., Wood, M. Object-Oriented Inspection in the Face of Delocalisation, appeared in *Proceedings of the 22nd International Conference on Software Engineering (ICSE) 2000*, pp. 467-476, June 2000.
5. Dunsmore, A. *Survey of Object-Oriented Defect Detection Approaches and Experience in Industry*, Technical Report – EFoCS-X-2000, July 2000.
6. Fagan, M. E. Advances in Software Inspections, *IEEE Transactions on Software Engineering*, Vol. 12, No. 7, July, 1986.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*, Addison-Wesley, 1994.
8. Gilb, T., Graham, D. *Software Inspection*, Addison-Wesley, 1993.
9. Laitenberger, O. *Cost-effective Detection of Software Defects through Perspective-based Inspections*, PhD Thesis, University of Kaiserslautern, Germany, 2000.
10. Linger, R. C., Mills, H. D., Witt, B. I., *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.
11. van Solingen, R., Berghout, E. *The Goal/Question/Metric Method*, McGraw-Hill, 1999.
12. Soloway, E. Pinto, J. Letovsky, S. Littman, D. and Lampert, R. Designing Documentation to Compensate for Delocalised Plans, *Communications of the ACM*, Vol. 31, No. 11, pp. 1259-1267, 1988.
13. Weller, E. F. Lessons from Three Years of Inspection Data, *IEEE Software*, Vol. 10, No. 5, pp. 38-45, September 1993.