Problems Pitfalls and Prospects for 00 Code Reviews

Marc Roper
Problems, Pitfalls and Prospects for OO Code Reviews
Marc Roper and Alastair Dunsmore

Department of Computer Science
University of Strathclyde
Livingstone Tower
Glasgow G1 1XH, U.K.

marc@cs.strath.ac.uk   apd@cs.strath.ac.uk
Introduction

• Software inspection well established as an effective defect-detection technique
• Developed prior to the OO boom - unclear what impact this has
• May result in “sub-standard” OO code inspections
Problems of Object-Oriented Code Inspection

- Structure - network of communicating objects (delocalised plans)
- Dynamic binding and polymorphism
- Inheritance
- Generics
- Chunking
An Inspection Experiment

- Aim to improve the process by investigating the nature of “hard to find” defects
- Student-based, using Java. Volume of material and time allowed similar to published industrial figures
- Necessary to create some defects - two sets of 10 defects seeded into around 200 loc.
Initial Defect Classification

Defect Classification

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

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

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

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

• Selection effects
• Learning curve
• Unrepresentative:
  – subjects
  – code
  – defects
  – process
Results

- Defect Detection and Comprehension
- Inspection Strategy
- Defects
Defect Detection and Comprehension

Subjective Measure

Number of Defects Discovered (out of 10)

Rsq = 0.3246

Subjective Measure

Number of Defects Discovered (out of 11)

Rsq = 0.0606
Inspection Strategy - Group A
Inspection Strategy - Group B

[Graph showing data distribution with box plots for different categories labeled DEFT1 to DEFT7]
Inspection Strategy - Detection Rate

Time (in minutes)

Average number of defects found

Group A
Group B
Defect Detection

- Analysed according to frequency of detection
Defect Detection

Described each defect using a number of key words and phrases:

- Locality (method, class, system)
- Algorithm/Computation
- Use of Class library
- Wrong object
- Wrong message
- Data flow error
- Method size (S,M,L)
- Instance variable misuse
- Omission
- Commission
- Inheritance / Implementation
- Override
- Diagram mismatch
- Domain knowledge
Defect Detection - Observations:

- Locality well mixed, but harder defects tend to have class locality
- Defects involving class libraries, wrong objects and messages tend to be harder to find
- Method sizes are mixed but no hard defects appeared in the small methods
- Defects involving inheritance, overriding and design mismatches tend to be hard to find unless there is supporting domain knowledge
Defect Detection - Rule Induction

- Analysed data using rule induction package:
  - Defects involving a domain knowledge clash instance variables have a very high probability of being found
  - Defects involving the use of the Java class library never appear in the easy range of defects.
  - Defects with a wrong message and which use the Java class library, on average also appear at least in the middle area for detection
Rule Induction Observations, continued...

- Locality, although sometimes varying significantly, does appear to make a difference when finding defects.
- Defects which have no domain knowledge clash but have diagram conflicts (i.e. involve inheritance, overriding, abstract classes etc.) have a less than 50% chance of being found.
Conclusions

• Results tend to agree with theoretical observations about OO difficulties
• “Difficult” defects tend to be those that span the class boundaries
• Serious implications for inspection strategy and possible support for process improvement
Prospects for Improvement

- Improved checklists
- Scenarios/Perspective Based Reading
- Visualisation
- Contextual Access
- Experience Base
- Chunking strategy
- Very keen to hear about experiences
Problems, Pitfalls and Prospects for OO Code Reviews

Marc Roper and Alastair Dunsmore

Department of Computer Science
University of Strathclyde
Livingstone Tower
Glasgow G1 1XH, U.K.

September 1999

E-mail: marc@cs.strath.ac.uk, apd@cs.strath.ac.uk

Abstract

Software inspection is a well-known technique used for defect detection. For the most part, code inspections have been based on procedural languages, e.g. C, with minimal interest in the effect object-oriented code potentially has in complicating the process, and what can be done to help an inspector overcome these problems. This paper describes an investigation into the OO inspection process that focuses on defects and their associated detection rates. It was found that several characteristics of OO defects, sending messages to objects, use of class libraries, data flow error, made certain defects more difficult to detect. The paper concludes by describing a selection of possible aids that could be used to help the inspector find the defects with these troublesome characteristics.
1 Introduction

The software inspection process was originally developed by Fagan [11] and has established itself as an effective means of finding defects. There exist many reports highlighting the success of inspection as a technique [10], [15], [25], however the vast majority of reports in the literature relate to inspections carried out with procedural languages, the predominant language used when inspections were originally created in the 70's. In the last decade or more, the Object-Oriented paradigm has appeared, made popular by C++, and now even more so by Java. In a survey carried out recently [9] there was an indication of the continued growth in the popularity of Java as a development language.

Currently there appears to be a lack of information regarding Object-Oriented code inspection, and how features of the OO language may make an impact on inspection techniques and the inspection aids that are currently used. This omission may be for several reasons: not many companies may be working with OO code (but still working on procedural code with inspections); it may be that OO code inspections are being carried out in the same way as traditional inspections have been; it may be that code inspection has been ignored as a technique for evaluating OO code, or that traditional techniques are seen as appropriate for OO code. This lack of information could lead to substandard inspections in relation to OO code, with the possibility of many OO specific defects going unnoticed. With the increased complexities of programming large systems with OO languages, and the possible costs involved with tracing and rectifying major OO defects, it is important that programmers are using the most effective and efficient techniques. To date, there has been minimal investigation or experimentation carried out into OO inspections, the impact of the OO paradigm, and its affect on any inspection support mechanism. This work is a first step in that direction, to investigate OO code inspections, and to determine if more needs to be done in this area.

2. Problems of Object-Oriented Code Inspection

The object-oriented paradigm has gained widespread acceptance [7] and has created many benefits for the programmer such as producing better structured and more reliable software for complex systems, greater reusability, more extensibility, and easy maintainability [19]. With these successes, there have also arisen new problems to be tackled. In 1994, Jones [17] listed some of the gaps in information about the object-oriented paradigm. One of those gaps was in the area of inspection. Jones noted that "Since formal inspections are the most effective known way of eliminating software defects, software quality assurance personnel are anxiously awaiting some kind of guidance and quantitative data on the use of inspections with object-oriented projects" [17].

According to Gamma et al. [13], the structure of an object-oriented program at runtime is vastly different to that of its code structure. They claim that the two structures are largely independent. Where the code structure is frozen at compile-time, the runtime structure consists of rapidly changing networks of communicating objects. This makes it very difficult to understand one from the other.
Dependencies exist in all code, but their number are increased in object-oriented languages [7], [29]. A dependency exists where if two entities in a system, say X and Y are related, then if X is modified, this could have possible repercussions on Y.

Dynamic binding involves not knowing the class of a particular object assigned to a variable, as this is only determined at run time [29], [21], [5], [30]. When a method invocation occurs, only at run time can an object's class be correctly identified. This also increases the complexity of the dependencies in a program.

Polymorphism involves naming consistency [29], [21], [5], [30]. The idea behind polymorphism is that different objects have a method with the same name, and if that method is invoked, the same basic action should occur, no matter the object. If method behaviour is not consistent throughout the program, then comprehension problems may arise.

Booch [5] describes inheritance as "a relationship among classes, wherein one class shares the structure or behaviour defined in one (single inheritance) or more (multiple inheritance) other classes". The complexity of the program comprehension process increases as some classes cannot be fully understood in isolation, and have to be looked at in conjunction with others.

The structural style of object-oriented programs differs from that of conventional programming languages [29]. Method sizes may be very small [30], and this could lead to a wide dispersal of code to perform a given task. This means traversing up and down the object inheritance hierarchy in an attempt to locate where the work is carried out.

A generic class acts as a template and can be instantiated by other classes, objects, etc. [21], [5], mainly acting as a container class. A simplistic example of a generic class is that of an array. An array can be defined to contain elements of any one class. Any subclasses of the defining class can also be placed into the array. This leads to the same problem that occurred with dynamic binding. Only at run time can the contents of a generic class be exactly determined.

From experience with inspections over the years, it has been found that there is a limit to the number of lines of code that can be read during the preparation stage, beyond which the number of defects found per thousand lines of code decreases. Fagan [12] suggests that a maximum of two, two-hour inspection sessions are the most that should be carried out each day. He goes on to suggest that good inspections are tied to thoroughness, and that no more than 125 non-commentary source statements per hour are read. Weller [28] confirms this from results taken from over 400 inspections. He found that less than 200 lines of code should be read per hour during the preparation process to get the maximum number of defects per thousand lines of code. It is believed that if the process is rushed, or too much code is looked at, then the effectiveness of the inspection process is greatly reduced [21], [28], [2]. These factors limit the amount of code that can be looked at during an inspection meeting.

In each of these object-oriented specific features, they depend and refer to other classes. A more organised method of splitting object-oriented code has to be found, as an arbitrary split is not good enough to allow an inspector to be able to completely
understand the code they are looking at. Macdonald [21] also highlights a similarity between inspection and testing, "although it is tempting to test a class in isolation, it must actually be tested in context of its parent classes because of the possibility of hidden interactions". Currently, few papers have been published which attempt to resolve this problem.

3 An Inspection Experiment

The area under investigation in this experiment involves the defects themselves: which caused the inspectors trouble, especially in relation to specific OO defects; is there a link between defects with a similar response rate, and what might be done to help find the hard defects.

Design

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

The experiment was based around a Java code inspection exercise. Since this investigation was solely concerned with the effort of the individual, no group component was carried out. The code inspection was paper based, no tool support was provided. During the inspection subjects used the ad-hoc style of inspection.

At the beginning of the class, subjects were split into 12 groups of approximately equal ability, based on marks from previous classes. 11 of the 12 groups contained 4 subjects, and the remaining one contained 3 subjects. These groups were used for earlier parts of the software engineering class. The same code was to be given to all subjects during the experiment, but to maximise the number of possible defects seeded within the code, two different sets of defects were created; A and B. Using the earlier subject grouping from the class, groups 1 to 6 were assigned to defect group A and groups 7 to 12 were assigned to defect group B.

Defects

When carrying out an experiment involving code inspection, a selection of defects is also required. Previous code inspections have predominantly been carried out on procedural code. There is very little material in the literature discussing OO code inspections, OO defects or any differences between procedural and OO inspections (if any actually exist). A search of the literature was carried out for known or possible problem areas in OO code to evaluate which would be the most appropriate defects to seed in the code. From the information found, a list was drawn up of all possible defects suggested for object-oriented code. Taking into account any overlap between defects, these were abstracted and narrowed down to a list of four groups of defect type. These four groups, along with illustrative sub-classifications are shown in Figure
1. It must be emphasised that this is only an approximate classification, and in many cases, defects can fall into one or more of the groups.

**Defect Classification**

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

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

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

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

---

**Figure 1. Defect Classification for OO code based on literature search**

For the experiment, the defects created fitted into the derived categorisation (Figure 1). Some defects were naturally occurring, e.g. were in the code when written by the course lecturer, others were seeded in the code based on information gathered in the literature search. Ten defects were present in the code given to group A subjects and ten defects were present in the code given to group B subjects. One defect was present in both groups, 3 others defects were similar in nature, but the syntax varied between the groups.

**Data Collection**

Data from the experiment was collected via a Defect Report form used by subjects to record defects found during the inspection task. This form was tested during the initial training phase of the experiment, and no major problems were found with it.

**Training Activities**

An introductory lecture and training phase was carried out before the experiment proper. The lecture lasted 50 minutes and introduced the basic premise behind inspections, their uses and problems. The next day a training session lasting 2 hours was held and was run informally to allow subjects to ask questions and overcome any conceptual problems about the inspection process. The experiment proper was held one week after the training exercise.
Conducting the Experiment

In the experiment proper, subjects were given up to a maximum of 2 hours to complete the inspection. Once all the subjects were seated in their two groupings, groups A and B, they were supplied with a booklet containing the inspection task material. If a subject went beyond the 2-hour limit for the inspection task, they would be asked to stop working.

Threats to Validity

An empirical study can be distorted by influences, which may affect the experimental variables without the knowledge of the researcher. This possibility should be minimised as much as possible. These included:

- Selection effects, which can occur through variations in the natural performance of individual subjects. As part of an earlier exercise in the class, the subjects had been split into 12 groups of roughly average ability.
- Plagiarism, which was not a concern as the experiment was carried out under exam conditions.
- Learning curve for the subjects associated with the programming language used; in this case Java. Prior to this class subjects had previously used the object-oriented languages of Eiffel and C++. To reduce any possible effect due to the use of a new language, earlier sections of the class had the subjects (in groups of 3 or 4) code a small program (approximately 8 pages in length). This was followed later by a more substantial system. The average number of classes created for this task was 21 (range of 13 - 40) and the average length of the systems was 2755 lines of code (range of 1200 - 4500). The ranges vary so much due to some groups implementing a full graphical interface. The code used for the inspection exercise proper was an extension of this system. During these pre-experiment programming exercises, subjects were working in their groups (1 to 12), where each group contained 4 subjects, bar one, which contained 3 subjects.
- There was no way to monitor the preparation of subjects prior to the experiment as they were working 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 experiment results.

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 could not be avoided due to time and resource constraints.
- The Java code may not be representative (in complexity or stylistically) of industrial software. In this case though the code inspected was part of a substantially larger 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 was mentioned earlier, a thorough search of the literature was carried out, the results of which were used to base decisions on types of appropriate defects.

• The inspection process used during the experiment may not have been representative of industrial software practice. This experiment focused only on the defect detection phase (for the individual), used the ad-hoc method for inspecting the code, and did not involve any presentational overview by the author, group collation phase or rework phase.

4 Results and Analysis

The following details the results and analysis of the experiment and is split into three separate sections. The first deals with the link between defect detection levels and comprehension, the second looks at the order and time information gathered from the experiment, exploring the ways in which the subjects carried out their inspections, and the third investigates the defects themselves.

4.1 Defect Detection and Comprehension

Although subjective questions have been criticised in the past as being unreliable [26], they are cheap to create and require only a small effort on behalf of the subject to answer. The question asked the subjects how much of the code they just inspected did they understand. The subjects were presented with five options: "Some of it", "Half of it", "More than half", "Nearly all of it" and "All of it". Of the 18 subjects in group A, 1 replied that they understood half of it, 10 replied that they understood nearly all of it, and the remaining 7 subjects replied that they understood all of it. Of the 23 subjects in group B, 1 replied that they understood half of it, 1 replied that they understood more than half, 15 replied that they understood nearly all of it, and the remaining 6 subjects replied that they understood all of it. Figure 2 shows the scatter plots of subjects subjective scoring against number of defects found during the inspection task for both groups A and B.

![Figure 2. Subjective score against number of defects found in inspection (left – group A, right – group B)](image-url)
The subjective question was measured on the ordinal scale (because the interval gap between each option is not the same). As a result of this, the correlation calculations were carried out using Spearman’s rho, to compare the subjective measure (measured on a scale from 1 to 5) results with the defect discovery results. The results are, as before, based on a two-tailed analysis.

For group A (shown on left side of Figure 2), the Spearman correlation produced an r value of 0.493 which is significant at the 0.05 (5%) level, meaning that there is a 1 in 20 probability that this is a chance result. The r value for group A is positive and this suggests that as subjects understand more of the code (represented by the subjective measure), there is an increase in the number of defects they discover. This indicated that for Goal 1, H₀ can be rejected and H₁ accepted. For group B (shown on right side of Figure 2), the Spearman correlation produced an r value of 0.404 which, although not significant at the 5% level is significant at the 10% level. The r value for group B is also positive, and there is 1 in 10 probability (10%) that this is a chance result. H₀ can be rejected for Goal 1, but only at the 10% level of significance.

4.2 Subjects' Inspection Strategy

This section looks at the time and order in which defects were discovered during the inspection phase of the experiment. When subjects noted a defect in the inspection phase, they also noted the time at which they found the defect. This was to allow a picture to be built up of the order in which defects were found and the time taken to find those defects. This would hopefully give an indication as to how subjects carried out their inspection of the code. Boxplots showing the timing information visually for each group can be seen in Figure 3 and Figure 4.
When looking at the discovery order of the first three defects (defects 8, 9 and 1) in group A it can be see that, in general these defects were found in order. Timing information for these three defects shows that most subjects found these defects within the first 20 minutes. It seems that at least during the beginning of their inspection, subjects began reading through the code in the order it was provided.

Beyond defect 1 there was a greater variation in subjects discovery order. What Figure 3 does indicate however is that defect 2 (already mentioned) and defect 4 were in general not found in their presentation order. The timing information tends to suggest that these defects were found upon subsequent re-reads of the code. Ignoring these defects shows that the remaining defects (3, 5, 7 and 6) were, for the most part, found in their presentation order. Subjects took approximately 40 minutes to go through the code at least once. The average inspection time for group A was 77 minutes.

Looking at Figure 4 for group B, it is much harder to see any clear picture. What can be said is that the bulk (80%) of the defects discovered by subjects were found within the first 50 minutes. For the same amount of time, group A subjects found 78% of all their defects. It is very surprising that most subjects managed to find such a high percentage of defects within the first 50 minutes. This could indicate that subjects performed well during the inspection, or it could suggest that the defects were, in some cases, too easy to find and not representative of realistic defects.

Of the three defects which stand out as being found later on during the inspection process, two were OO related (i.e. the incorrect implementation of an interface and
the missing method call leading to the wrong object being passed to another method) and one defect not OO related, (i.e. the incorrect placing of a method call). This last point hints that at least some misplaced method calls which are not immediately obvious as a defect do require a greater understanding of the whole system to appreciate (bearing in mind that only 6 out of 18 subjects actually managed to find this defect).

Figure 5. Graph of average subject performance for groups A and B

Figure 5 shows the average number of defects found by subjects within the two hours allowed for the inspection task. It shows that the performance of both groups A and B was reasonably similar. This indicates that the balance of defects between both groups was similar. What can also be seen in Figure 5 is that beyond the 60-minute mark, there was an average of only 0.5 defects found per subject. This indicates that for this experiment, working beyond the 60-minute mark brought dramatically diminishing returns.

4.3 Detection of Object-Oriented Defects

Many of the defects seeded in the code were OO in nature, involving aspects unique to OO languages and systems. There is currently a lack of research on OO inspection, and it is difficult to tell if OO languages will have an impact on the way code is inspected, or the aids used to help this process. This section begins by examining the defect detection results of the inspection part of the experiment, grouping defects by the initial defect classification scheme discussed earlier (see Figure 1). Following this, a closer look is taken at the defects, discussing which were harder to find and which were easier to find (also checking to see if the defects with OO features were hard to find), in an attempt to find if there are any links or common features shared between defects with a similar discovery rate. This work leads to a new classification for the
defects, based on the attributes and features of each defect. This then leads to the final part of this section with comments on what aids/techniques could be used to help with the difficult defects.

To help find possible links between defect type and ease of detection, it was first necessary to regroup the defects. To do this the initial classification scheme was dropped and all defects were grouped together (from groups A and B). This is represented in Figure 6, which shows the percentage of subjects (y-axis) who found each particular defect (x-axis) and which group the defect belonged to (the colour of the bar).

What cannot be seen very well in Figure 6 is what, if any are the common features between defects with a similar response rate. Since the initial classification scheme was approximate and lacked detail, a fresh look was taken at each defect. For each defect, a series of key words were compiled, reflecting features associated with the defect. Following this, an attempt was made to group the defects together based on their key descriptors. This suggested that there was four main groupings of defects: inheritance/override, data (usage and miss-usage), instance variables, and algorithmic. To further investigate defects with a similar response rate, the defects were listed in response order (percentage) along with their key descriptors.
<table>
<thead>
<tr>
<th>%</th>
<th>Domain knowledge</th>
<th>Diagram mismatch</th>
<th>Override</th>
<th>Inheritance/Implementation</th>
<th>Commission</th>
<th>Omission</th>
<th>Instance variable misuse</th>
<th>Method size (S, M, L)</th>
<th>Data flow error</th>
<th>Wrong message</th>
<th>Wrong object</th>
<th>Use of class library</th>
<th>Algorithm/computation</th>
<th>Locality (M, C, S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>8A</td>
</tr>
<tr>
<td>100</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>9A</td>
</tr>
<tr>
<td>94</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>1A</td>
</tr>
<tr>
<td>91</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>7B</td>
</tr>
<tr>
<td>89</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>5A</td>
</tr>
<tr>
<td>87</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>1B</td>
</tr>
<tr>
<td>87</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>8B</td>
</tr>
<tr>
<td>83</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>4B</td>
</tr>
<tr>
<td>74</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>2B</td>
</tr>
<tr>
<td>73</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>5B</td>
</tr>
<tr>
<td>67</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>L</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>6B</td>
</tr>
<tr>
<td>67</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>L</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>4A</td>
</tr>
<tr>
<td>64</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>3A</td>
</tr>
<tr>
<td>63</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>4A</td>
</tr>
<tr>
<td>50</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>2A</td>
</tr>
<tr>
<td>50</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>3A</td>
</tr>
<tr>
<td>48</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>6A</td>
</tr>
<tr>
<td>48</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>10A</td>
</tr>
<tr>
<td>42</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>10B</td>
</tr>
</tbody>
</table>

Table 1. Defects described by key words

Table Notes:
Locality - (M)ethod, (C)lass, (S)ystem
Method size - (S) 0-4, (M) 5-10, (L) 11+, (-) No size available
From looking at Table 1, the following points can be observed:

- Locality of defects is fairly well mixed, with the only class locality defect appearing in amongst the hard defects.
- Defects involving the use of the Java class library, wrong object and wrong message all appear in the right half of the table, indicating that none of the easy defects have these characteristics.
- Data flow defects are mainly found in the middle of the table, indicating that they fall between easy defects and hard defects to find.
- Method sizes are fairly well mixed, the only point to note is that no defects below those with an 83% discovery rate involved small methods, only medium and large sized ones.
- For the two inheritance defects (8A and 9B) there is one difference in their key descriptors. Defect 8A involved the use of domain knowledge, whereas defect 9B did not. The inheritance in defect 8A involved another class, which was part of the system but directly conflicted with the knowledge held by the subjects. Defect 9B involved the incorrect implementation of an interface class, which is defined by the Java language and since this was more of a language issue (rather than conceptual) defect, it did not seem to cause a conflict with subjects domain knowledge, making it harder for them to spot the defect.
- For the two override defects (7B and 6A) there are several differences in their key descriptors. Defect 7B was one of commission (superfluous code), whereas defect 6A was one of omission (missing code - a class method should have been overridden). In defect 7B, the method involved is small, whereas for defect 6A there is no method size (due to the missing method override). Finally, defect 7B involved domain knowledge, whereas defect 6A did not. This was due to defect 7B using a method called `isBorrowable`, which returned the value `false`. Subjects were able to tell that this was wrong from their domain knowledge (i.e. that they should be able to borrow a video). All the subjects in the experiment reported this defect as an incorrect return value, either not noticing or not reporting the incorrect override issue. Defect 6A was more subtle as the method missing from the `Video` class already appeared in the class diagram further up in the class hierarchy, and it was not immediately obvious that the method needed to be redefined in `Video` (apart from its appearance in the class hierarchy diagram).

To investigate the information shown in Table 1 further, it was entered into C5.0 [6], a rule induction system. A rule induction system attempts to draw out a series of rules (or patterns) from information supplied to it. All the information shown in Table 1 apart from the percentage values were entered into the rule induction system. Instead of entering the exact percentage value, defects were grouped together and given textual references to stop fragmentation of the data. For this run, 4 percentage groups were chosen, 100 - 83 % : VHIGH (8 defects found by almost everyone), 74 - 67 % : HIGH (4 defects which start to show some problems in detection), 50 - 33 % : MIDDLE (6 defects which are found by half or less of the subjects, proving more difficult to detect) and finally 0 - 0 % : LOW (the remaining 2 defects which were found by none of the subjects). This is very much a subjective split, and this aspect was modified on subsequent re-runs of the rule induction system to investigate the effect of changing the boundary of the percentage groups (discussed later).
There are several general trends seen in Table 1, which are confirmed by the information provided by the rule induction data.

- Defects involving a domain knowledge clash with subjects knowledge or defects involving instance variables have a very high probability of being found and are not difficult defects to find.
- Defects involving the use of the Java class library never appear in the easy range of defects.
- Wrong message defects can vary between high and low detection, but are on average found within the middle ground for detection (e.g. around the 50% area).
- Those defects with a wrong message and which use the Java class library, on average also appear in the middle area for detection.
- Locality, although sometimes varying significantly, does appear to make a difference when finding defects. Defects involving class locality consistently fall under 50% (detection rate), those at the method level in general fall into the 67-87% bracket and the system defects vary wildly from low detection rates to those in the 80's and above.
- Defects which have no domain knowledge clash but have diagram conflicts have a less than 50% chance of being found (this relates to the implementation and override defects).

What has been found is that there is an indication that some of the defects that have an OO feature, e.g. implementation and override (when lacking in domain knowledge) and message passing did cause problems for subjects. Defects that involved accessing the Java class library also appeared to give problems, indicating that subjects could have made assumptions about how classes in the library operate, or perhaps did not perceive a need to access it. Defects involving instance variables or those which provided a domain knowledge clash did not appear to present many problems for the vast majority of subjects. Although not considered an important factor by the rule induction tool, it can be seen that none of the defects with a low discovery level are present in small methods. This suggests that defects residing within small methods (1-4 lines of code) are relatively easy to find. Also not picked by the rule induction tool as an influencing factor was data flow errors. Although one of the defects involving a data flow error received a 94% discovery level, the other 5 defects involving data were not as easy to find. Data flow errors fall into a middle category, neither hard nor easy to find, indicating that these defects cannot be ignored.

Using several different methods, an analysis has been carried out on the defects used in this experiment, detailing key descriptors for each defect and investigating what made certain defects difficult to find. Various defect attributes, e.g. override, inheritance/implementation, commission, omission, method size, data flow, wrong object were found by the use of a rule induction tool to have minimal effect on the difficulty of detecting defects, but defects involving use of a class library, message passing, a lack of domain knowledge clash, as well as the locality of the defect were all found to have a measurable impact.
6 Conclusions

This paper detailed the design and execution of an experiment investigating whether OO related defects prove difficult to find. The purpose is to identify areas of possible support for the OO inspection process in order to assist in the detection of "hard-to-find" defects. The majority of subjects performed their inspection by reading the code in the order it was presented for them. Subjects did not seem to follow method calls between classes. Several defects in groups A and B were found by subjects out of presentation order, well into the inspection task. These defects seemed to require a greater understanding of all the code to be inspected before they could be detected. About 80% of all the defects detected by subjects in both groups were detected during the first 60 minutes. Beyond the 60-minute mark, there was very little return for effort expended. Defects which were difficult to find, were generally found by subjects who had performed well during the inspection.

Through the creation of a key word classification index for each defect and utilising a rule induction system, the following points were noted:

- Java interface usage (implementation) and method override defects not involving domain knowledge clash were among the more difficult to find defects.
- Defects involving incorrect messages sent to objects (especially if involving the Java class library) proved harder to find.
- Defects involving links to the Java class library proved difficult to find.
- None of the difficult defects appeared in small methods (1-4 lines of code).
- Defects involving instance variables or those which provided a domain knowledge clash did not appear to present any major problems for the majority of subjects.
- Data flow defects were neither easy or hard to detect.

The defects used in the experiment were based roughly on potential problem areas suggested by a search of the literature, but what about more recent developments, especially within the Java programming language - how will these affect inspection? Will they introduce new problems and, e.g. inner classes, large class libraries, etc.

7 Prospects

This final section briefly details some possible ways to support the inspector during the inspection of OO code, making it easier for them to locate the types of problem defect highlighted previously.

Improved checklists (specifically for OO and Java)

One solution to the problem is to modify current checklists to take into account some of the areas that have proved problematic. This in itself is not a simple task as the language used can mean great differences. Humphrey [16], as part of his discussion on inspections, presented a C++ checklist. Several of the areas covered by the checklist; dynamic storage allocation and pointers are specific for C++, whereas Java
as an OO language does not have pointers or low level memory allocation (e.g. using alloc or malloc) but instead any low level dynamic memory allocation is done behind the scenes by Java.

This experiment has shown that the Java class library plays an important part in the difficulty of defects, and although the Java class library is far larger than the small library of functions provided with C++, there currently appears to be no series of checklist questions which cover this aspect of programming languages. Class libraries are gaining more and more popularity, especially with the advent of Java. Why write a class or function when someone else has already done it for you. The problem arises in how the libraries are used, and if they are used correctly.

Other points which might benefit from being expanded from their current form in C++ checklists are checks on messages to objects (i.e. right/wrong message to the right/wrong object) and the use of inheritance/implementations (i.e. frameworks).

This option may be the cheapest option to implement for those who already use checklists but wish to use them on OO languages, requiring no new techniques, training, or tools. There is however a potential problem. An inherent feature of checklists is that they guide (force) an inspector to jump around a program, only considering a small fragment of code each time. If comprehension is indeed important to the inspection of OO code, checklists could be seen to hinder this process, and not allow inspectors to generate a full understanding of the code under inspection. More research needs to be done into the usefulness of checklists with OO code inspections, to investigate their usefulness in OO code inspections, and to see if they hinder an inspector's ability to comprehend the code.

Scenarios/Perspective based reading

Scenarios, like the checklist, are an aid (or guide) during the individual phase of a code inspection. The scenario technique was created by Porter et al. [24] to address a perceived lack of effectiveness in the use of ad-hoc and checklist methods. Porter et al. describe a scenario as a "collection of procedures that operationalise strategies for detecting particular classes of defects". Each inspector is then given one scenario, each of which is different from the rest of the inspectors on the team. Multiple inspectors are required to obtain a reasonable level of coverage of the document. Currently, scenarios are available only for software requirement specifications; none have been produced for code documents. More recently, perspective based reading (PBR) has been put forward as an inspection technique [1]. PBR is similar to scenarios, but it offers a more structured and well-defined process than scenarios, inspecting from different, well-documented perspectives/viewpoints. As with scenarios, PBR were initially used with software requirement specifications, but have since been adapted for code inspections [20], although further experimentation will be required to verify their effectiveness in this area.

It could be that one of the perspectives used to inspect the code could be tailored to look specifically at such OO problems as have already been mentioned. This, like the checklist however would have to be tailored on a per language basis, with some core OO elements always present. However, as well as perhaps having an OO code
specific perspective, other perspectives would also have to be altered, e.g. tester and analyst perspectives, to take into account any OO idiosyncrasies and differences. For any organisations already running some form of scenario/PBR technique for inspection, it would involve the incorporation of new perspective(s) and training in them if they proved to be substantially different from other perspectives currently employed.

**Visualisation**

In the last decade or so there has been a steady increase in the number of reverse engineering tools developed with a variety of visualisation features. Reverse engineering involves the analysis of a system to identify its components and interrelationships, creating representations of the system in another form or at higher levels of abstraction [8]. The idea behind using visualisation is to try and reduce the time spent comprehending a program during software maintenance. It is said that up to 60% of the time spent on software maintenance is spent on comprehending the program.

Visualisations offer many possibilities: effective visual interfaces allow interaction with large volumes of data in a fast and effective manner [14], they can be more memorable, more accessible and faster to grasp than basic text [23], and help lower the interpretation load placed on a human by providing visual displays for abstract chunks of textual structures [31]. Of special note is that Petre et al. [23] concluded that currently while there are still some people who prefer to read paper copies of code when carrying out comprehension and debugging, this should change with the continually increasing popularity of OO languages, which are non-linear and therefore make the code more difficult to follow.

Visualisations are useful when displaying class hierarchies, lists of methods in classes (including methods inherited from other classes), the relationship between a class and the rest of a system, etc. They could also be used to better relate code to class libraries.

**Contextual Access**

Contextual access is the use of hypertext links (as commonly used in web pages) to access desired information from the location of the problem. This provides access to desired information from within the context that it was required, e.g. reading some code and finding a call to a class method in another class. Contextual access using hyperlinks could allow the selection of the method call which could then allow a user to view the code for the method, or its specification, or just a textual description of its purpose.

Hypertext access has been popularised as the standard interface for accessing web pages on the World Wide Web (WWW). Since then hypertext has been used for a variety of other areas including decision support systems [4] and knowledge based systems (KBS) [18]. More recently Mao and Benbasat [22] carried out an experiment investigating the use of hypertext links as contextual access to explanatory knowledge.
in a KBS. Their results showed that the larger the disparity between the user's knowledge and the expertise contained within the KBS, the greater the need for contextualised access to explanatory knowledge. They also showed that knowledge has to be readily accessible for it to be used. If it is available, but not directly, then it is less likely that an effort will be made to find it. The convenient access to information within the context of problem solving appears to be a key aspect to a successful system using hypertext links.

One such tool that, to some extent, encompasses both visualisation and contextual access is SNiFF+ [27]. Its uses visualisation to display a class hierarchy diagram, and lists of methods (including those inherited) can be displayed for a particular class with graphical symbols used to describe their accessibility (e.g. public, private, etc) as well as showing if the method is overriding an inherited method. Selecting classes in these diagrams will bring up another window containing the actual code of the class (or method) selected. Hyperlinks are also available in the code window, highlighting method names and class names, allowing you to display these items if selected (although in each case a new window is displayed, which can eventually clutter up a display).

Contextual access could be used to quickly access class libraries, cutting down the need for inspectors to waste time searching through various help systems and manuals. It could also be used to link the code to design documents and diagrams, as well as perhaps encouraging subjects to follow method calls, rather than read all the code in order.

If inspectors were to inspect their code within a similar kind of environment as the one described, access to information in class libraries, help files, specifications and class diagrams could be only a single mouse click away, improving an inspectors efficiency and overall understanding of the code.

**Experience Base (built up from previous inspections)**

The idea behind an experience base is to accumulate experiences that can be used to improve the software process and product [3]. Lessons learned from defect detection techniques, defect models, as well as project specific lessons can be recorded. The experience base supports learning and reuse and can generate a tangible corporate asset in the form of packaged experiences.

An experience base could be used to store experiences with difficult defects, along with the characteristics that made it difficult, and how the defect was discovered. Experience of the removing elements from a vector problem (see defect 10 - group A or B) could be stored in an experience base for future reference, and through the use of a tool, could be brought to the attention of inspectors in future occasions.

In some respects, the checklist is a basic version of an experience base as it contains a series of pointers towards specific areas of code, which have been shown to contain defects in the past. Over time, checklists should be amended to reflect different types of projects and problems. A more advanced on-line version of this could be created, and made more accessible through the use of hypertext links or as part of a system.
such as was described earlier in this section. Such a system could allow contextual access to hints and information about specific problems and defects discovered from previous inspections, highlighting potential problem spots and speeding up the defect discovery process.

References


7. X. Chen, W. Tsai, and H. Huang, Omega - an Integrated Environment for C++ Program Maintenance, International Conference on Software Maintenance, November 4-8, 1996.


Marc Roper

Marc Roper received the BSc degree in Computer Science from the University of Reading in 1982 and the PhD degree in Computer Science from the Council of National Academic Awards in 1988.

He is currently a Senior Lecturer in Computer Science at the University of Strathclyde, Glasgow, Scotland. His research interests include the empirical evaluation of software engineering technologies, software testing and software inspections. He has published widely in all these areas. He presently serves on the editorial board for Software Testing, Verification, and Reliability and for Information and Software Technology.