On the improvement of a fault classification scheme with implications for white-box testing

Shimul Kumar Nath, Robert Merkel† and Man Fai Lau
Faculty of Information and Communication Technologies
Swinburne University of Technology, Hawthorn, Australia
†School of Information Technology, Monash University, Clayton, Australia
snath@swin.edu.au, robert.merkel@monash.edu, elau@swin.edu.au

ABSTRACT
Different testing techniques can be more or less effective on different fault types; therefore, testing methods that are most likely to detect the most common fault types should be preferred. However, to enable such a selection of testing methods, a suitable and effective fault classification scheme is essential. Much software testing research proposes techniques to generate test cases and evaluates these techniques based on some hypothesized fault classification scheme. However, there is a lack in the justification of whether such ‘hypothesized faults’ are realistic and how often these faults occur. Recently, Pan et al. analyzed the syntactic changes in the source code, made in fixing faults, of 7 open source software projects implemented in Java. Based on their experience, they proposed a fault classification scheme with relative frequencies of each fault type. As always, readers may question whether the resulting fault classification is reasonable and appropriate. Hence, we applied their method to Checkstyle, another open source Java program, as the subject of our study, hoping to validate the appropriateness of their fault classification scheme, with particular application for selecting testing methods. While we generally found their classification scheme reasonable, we also noted that some faults could be classified in multiple ways. We also found that the frequencies of fault categories in Checkstyle were significantly different to the seven systems studied by Pan et al., which had all shown to have quite similar frequencies. We identified several potential improvements to their classification, permitting the classification of a larger proportion of faults. We have identified new implications of the combined results for white-box testing, and proposed follow-up studies to examine these issues in more detail.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: testing and debugging

General Terms
verification, reliability

Keywords
bug patterns, bug analysis, fault classification, fault based testing, white-box testing

1. INTRODUCTION
The effectiveness of different types of software verification techniques is dependent on the nature of the faults present in the software. Most obviously, static analysis [9] checks source code for the presence of certain syntactic patterns that often indicate a fault - while these tools can reveal many software faults, they will only ever be able to reveal a subset of all faults present in software. Testing techniques, too, are differentially effective on different fault types. For instance, code coverage based testing techniques are less effective at revealing faults relating to missing cases (for instance, in a switch statement). Fault based testing [14] is a technique where test cases are specifically chosen to reveal whether a certain fault class is present in the software under test.

While the faults in any system under test are by definition unknown before testing, historical information about fault types that are likely to be present in the software under test will allow testers to choose testing methods that reveal a greater proportion of the faults present in the software. From a research point of view, information about the relative frequency of different types of faults will allow researchers to recommend suitable testing methods or develop improved testing methods. This testing selection strategy can be effective at detecting the more common fault types.

Hayes [7] used a number of fault classification studies in analysing the merits of various testing techniques for object-oriented software. However, no relative frequency information was considered (only a taxonomy was presented), and the author stated that their recommendations were based "...largely the author’s personal testing experience" rather than from the relative frequency. White and Cohen [19] justified the relevance of a testing technique for linear predicates by an empirical study by analyzing predicates in a set of “typical data processing programs”, finding that the vast majority of such predicates were indeed linear. The use of relative fault frequency to justify fault specific testing techniques remains something of a rarity. If such fault frequencies are to be used, however, it is important to select an appropriate and robust classification scheme.

Several fault classification schemes have been devised for
different purposes such as testing, debugging, and better understanding of bugs. Notably, Knuth [12] developed a classification scheme, based on the extensive records kept by the developer (Knuth himself) of the errors found in the typesetting software, \texttt{TeX}. This classification is inherently manual and difficult to replicate - few developers keep similarly extensive records of errors. An alternative approach have been devised by DeMillo and Mathur [5], based on the syntactic analysis of “bug fix” patches - the code changes made to fix identified faults. The goal of their study was to improve testing and debugging. However, their reported results were restricted to \texttt{TeX}.

A contemporary example of a syntactic fault classification is that of Pan et al. [18] who developed a classification scheme and a checking tool for Java software, and conducted an analysis using their tool, of the change repositories of seven well-known Java open source software projects. A key result of Pan et al.’s empirical study was across six of the seven systems analyzed, the frequency of the different patterns was relatively similar. If this result is valid across Java software more generally, it has important implications for software testing practice, particularly module-level white-box testing. To maximise fault detection capabilities, testing should be primarily aimed at revealing faults that fall into the most common classifications. As testing researchers, we are obviously very interested in improving testing practice based on such information. However, before investing research effort into such a project, we wished to answer several key questions:

- Is the classification scheme sound? Is it possible to replicate the fault classifications in other software?
- Does their claim that the proportions of faults falling into the various categories is similar for different projects hold for other software?
- Only about half of all bug fixes were classified. Can the scheme be modified to usefully classify a higher proportion of bug fixes?
- Are there other ways that their classification can be modified to provide additional information specifically for analyzing testing methods?

In this study, we use Pan et al.’s [18] classification scheme to manually classify the bug fixes in Checkstyle, an open source software project, to answer the above questions. We examine the soundness of the classification scheme, whether the proportions of faults classified into the classifications are in comparable proportions to Pan’s. We found their classification is incomplete and not orthogonal (some faults can be classified into multiple categories). The syntactic way is followed for the classification is providing scope to analyze deeper into the bug fix patterns to imply it on testing. We propose a number of improvements and extensions to their classification scheme based on our observations. In addition, we consider the application of the knowledge gained for testing, specifically fault based testing [14].

Section 2 discusses related work. Section 3 describes how we collected fault classification data. Section 4 reports our findings. Section 5 describes a number of issues identified with the classification scheme, and Section 6 analyzes threats to validity of the results. We discuss the implications of our findings in Section 7, including potential future work, and provide a concluding summary in Section 8.

2. RELATED WORK

Before discussing related work, it is important, here, to distinguish between the various terms used to describe misbehaving software. We follow the definition of the IEEE recommended practice [2], which describes a fault as

\begin{itemize}
  \item (A) A defect in the code that can be the cause of one or more failures. (B) An accidental condition that causes a functional unit to fail to perform its required function. A fault is synonymous with a bug. A fault is an error that should be fixed with a software design change.
\end{itemize}

As such, faults are the manifestation of mistakes in the actual software code. Failures describe the deviation from normal operation that may occur when the code containing a fault is executed. Finally, errors (sense B in the IEEE definition) are the human actions that caused the software to contain a fault.

Efforts to analyze and categorize software faults have a long history, and a number of different general approaches have been tried. Some authors [10, 15] have examined the connection between file-level or class-level statistics, such as module size or class cohesion, and fault frequency. This analysis is automated and uses unambiguous metrics. As such, it is straightforward to apply both in a research and, potentially, in an industrial context. However, from a tester’s point of view, while it is useful to indicate which modules, files or classes to test most intensively, it does not provide much indication of how to test those modules. Recently, some bug prediction research [17, 11, 16] proposed techniques to identify the most fault prone files and modules in a software project. These types of prediction techniques are significant in reducing the testing effort by predicting the most fault prone files or modules. However, again, they provide no indication of how best to test those files or modules.

An alternative approach was that of Knuth [12], who, rather than examining faults, directly analyzed the errors (in the IEEE sense of the word) found or reported in ten years of development of his well-known typesetting software, \texttt{TeX} into nine categories (Knuth also recorded seven categories of “enhancements”). \texttt{TeX} is a somewhat unusual software system, having been developed by one programmer (Knuth himself) over a ten-year period (as of publication of the paper). The error classifications are based on his own notes during development. For instance, the category “B” is described as follows:

B, a blunder or botch. Here I knew what I ought to do, but I wrote something else that was syntactically correct – a sort of mental typo. For example, in error no. 126 I wrote ‘before’ when I meant ‘after’ and vice versa. I was thinking so much of the Big Picture that I did not have enough brainpower left to get the small details right.

As this passage shows, this kind of error classification is inherently manual and would be very difficult to replicate for another large-scale system. It only works if the programmer who fixes the error is the same programmer who made the error, that programmer takes the time to reflect on their errors, and manually (and correctly) records the categorization for every error occurrence description. It represents a
valuable analysis of the thought processes responsible for a large sample of software errors (albeit by a programmer whose thought processes may not be the same as mere mortals), but the categorizations are not really precise enough to provide detailed guidance for testers - for instance, the “blunders” category does not really indicate what kinds of blunders were most common, nor whether there are particular programming constructs where they are likely to occur. This is particularly the case for analysis of white-box testing techniques.

Eisenstadt [6] asked professional developers to describe the “worst bugs” they had found, analysing a total of 59 reports of such bugs. This study is notable in that as well as categorizing the underlying causes of bugs, it reports reasons why these bugs were often very difficult to find. This study uses the term “bugs”, but the final categorization blurs concepts that are syntactic and thus probably relate more to faults, such as “wrong variable or operator” with others that speak more to the cognitive error made, such as “Language semantics ambiguous or misunderstood”. The categorizations were quite broad (with underlying causes categorized as, for instance “des.logic”, indicating “an unanticipated case faulty design logic”), and thus are not much help in choosing detailed testing techniques, particularly white-box testing techniques. The most common cause of “worst bugs” was the “mem” category, relating to problems with memory allocation, a much less severe problem in Java programs than languages requiring manual memory management.

DeMillo and Mathur [5] devised a syntactic classification scheme to strengthen debugging and choosing suitable testing. They constructed an automated classification tool and analysed the faults in Knuth’s study according to their syntactic classifications. Their classification scheme was largely based on the syntactic element of the code that was changed in a bug fix. For example, one fault category was “incorrect identifier” faults, where the bug fix involves replacing the incorrect identifier (variable, structure member, or method name) with the correct one. While not perfect, their tool successfully classified faults in a systematic, automated way. They also proposed to evaluate the effectiveness of various testing methods on the different fault categories distinguished in their classifications. Their reported results were restricted to one software artifact, Tex.

Static checking tools, by their very nature, automatically syntactically classify possible faults based on which of their rules are triggered by particular code fragments. FindBugs [9], for instance, can be used to generate statistical reports about the bug patterns found in a particular code base. In our view, such problems should be fixed by the use of a tool like FindBugs before testing is conducted, and therefore we are less interested in classifying them!

Pan et al. [18]’s syntactic classification scheme shared many characteristics of other classification schemes (such as DeMillo and Mathur [5]), but concentrates more on what the bug was. For instance, a high-level category in Pan et al.’s scheme is “If-related”, which encompasses all faults which were fixed by changing code relating an if statement. This was then subcategorized according to the nature of the change to this code. In the case of “If-related” changes, examples of subcategories included the addition of else branches, and the changing of an if condition expression, amongst others. This classification scheme is easily replicable on the bug fix data collected from Software Configuration Management (SCM) systems such as CVS or Subversion.

3. METHOD

3.1 Software artifact studied

For this study, we selected Checkstyle [3], a development tool which (coincidentally) is a tool to check that Java code adheres to a coding standard. It is a widely-used software tool, is itself implemented in Java, and has a well-maintained, publicly accessible CVS repository with a long history of source code changes.

3.2 Experimental procedure

Figure 1 summarises the workflow for analysing the faults in Checkstyle.

3.2.1 Extraction of the log file

The CVS (Concurrent Version System), a popular free version control system in software development, is used to keep track of all software changes. A centralized CVS server stores the change information in the repository which developers access using the appropriate client. A set of changes to various files is added to the repository by the developers as a “commit”, and is usually accompanied by a textual comment. The changes are recorded, the comments are logged, numbered, and permanently retained. As such, every revision committed to the repository can subsequently be retrieved. We used the TortoiseCVS client [1] to check out the “Checkstyle” open source software, extract the logs, and bug fix hunks (see Section 3.2.3 for definition).

In 2007, Checkstyle migrated its change repository to the newer Subversion version control system. However, for expediency, we used only the archived CVS repository. The Checkstyle CVS repository contained 6522 revisions to Java source files, and extended over several public release cycles. Therefore, the CVS repository was considered to be of ample size for this initial pilot study.

3.2.2 Extraction of the fix revision

CVS logs consist of a sequence of log messages written by the relevant developer to describe a commit to the repository. In an actively developed software project such as Checkstyle, commits are made for the purposes of enhancement or extension, as well as for bug rectification. As we are interested in analyzing bug fix patterns, we required a method to identify commits that were performed for fixing bugs. We used an approach initially developed by Mockus and Votta [13] to identify changes made for fixing faults.
We developed an extractor which searched for the keywords “fix”, “bug” or “patch” in the CVS log. Revisions with a log message containing at least one of these keywords were considered to be bug fix revisions. For those files changed in a bug fix revision, we noted the revision number of each relevant file, as well as the immediately preceding revision, for that file. Revisions to files other than Java source files were ignored. We considered the first 6522 revisions in Java files of Checkstyle; among them 1631 revisions in 476 commits were identified as bug fix revisions and extracted by our tool. From these, we randomly selected 100 commits with 373 bug revisions among the first 476 commits with 1631 bug revisions of Java files.

3.2.3 Classification of bug fix hunk pairs

We attempted to follow the method of Pan et al. [18] as closely as possible, with the exception of manually checking fix hunk pairs rather than using an automated tool.

Our extractor identified fix revision numbers and file names containing bug fixes. By using the file name and fix revision number, we extracted bug fix hunk pairs - the subset of lines in the two revisions of the file corresponding to what was changed - in “diff” format, by using “cvs diff” command through the Checkstyle project’s CVS server. After obtaining the fix hunk pairs, we manually classified the fix hunks based on Pan’s classification. We did not classify hunks containing changes only in comments, or hunks that contained many changes scattered at intervals through the hunk. We also ignored any hunk longer than seven lines.

Any fix hunk that could be classified using more than one pattern was recorded as unclassified. See Section 5.1 for additional discussion.

4. RESULTS

In our investigation, we found a total of 555 change hunk pairs (some revisions feature more than one change hunk pair), with 419 effective hunk pairs (that is, hunk pairs which contain some change in Java code). We calculated the proportion of commits and files that contained at least one classifiable hunk pair, and the proportion of hunks that were classifiable, and present this in Table 1.

We classified effective fix hunk pairs into the twenty-seven bug fix patterns, in nine groups, as in Pan et al. [18]. Table 2 shows the proportions of these fix hunk pairs in each of the 27 patterns.

4.1 Analysis of results

From Table 1, we found that the hunk, file and commit coverage are quite similar to Pan et al. [18]. The file coverage and hunk coverage are within the range reported by them. The commit coverage in our result is just under 89%, slightly less than their range (that is, 91-96%). Subjectively, these appear to be similar enough so as not to invalidate further comparisons.

### Table 1: Comparison of File, Hunk and Commit Coverage

<table>
<thead>
<tr>
<th>Coverage type</th>
<th>Checkstyle</th>
<th>Pan et al. [18] (range)</th>
</tr>
</thead>
<tbody>
<tr>
<td>File coverage</td>
<td>59.8%</td>
<td>53 - 75%</td>
</tr>
<tr>
<td>Hunk Coverage</td>
<td>52.3%</td>
<td>46 - 64%</td>
</tr>
<tr>
<td>Commit Coverage</td>
<td>88.8%</td>
<td>91 - 96%</td>
</tr>
</tbody>
</table>

From Table 2, we found that the percentage of major nine groups of fix patterns had some similarity to Pan’s result. For the If related (IF), Switch related (SW), Try/Catch related (TY), Loop related (LP) and Sequence related (SQ) fix pattern groups, our results for CheckStyle are within the ranges reported by them. However, the proportions for Class field related (CF) fixes were somewhat outside their ranges, and large differences were found in the Method call related (MC), Assignment related (AS) and Method Declaration related (MD) groups, with 12.4%, 0.9% and 34.3% respectively in our results, with 21.9-22.9%, 6.3-14.2% and 8.1-22.7% respectively in theirs.

However, this suggests that the differences between our results for the CheckStyle artifact, and Pan et al.’s results, are much larger than the differences within the seven artifacts of Pan et al.. We then computed the Pearson correlations for the category proportions of each of the seven artifacts and the Checkstyle, and tested the significance of these correlations using the Holm-Bonferroni method [8]. Table 3 shows the corresponding correlation coefficients and their statistical significances, with an overall \( \alpha \) (type 1 error) = 0.05. Note that the Pearson R correlations are comparatively small (ranging from 0.262 to 0.615) indicating that the frequency of bug patterns in Checkstyle is quite different to the seven artifacts studied by Pan et al. Furthermore, in three cases (ArgoUML, Scarab and MegaMek), the correlations were not statistically significant (that is, we could not reject the null hypothesis that \( r = 0 \)).

From Table 3, we found that the percentage of major nine groups of fix patterns had some similarity to Pan’s result. For the If related (IF), Switch related (SW), Try/Catch related (TY), Loop related (LP) and Sequence related (SQ) fix pattern groups, our results for CheckStyle are within the ranges reported by them. However, the proportions for Class field related (CF) fixes were somewhat outside their ranges, and large differences were found in the Method call related (MC), Assignment related (AS) and Method Declaration related (MD) groups, with 12.4%, 0.9% and 34.3% respectively in our results, with 21.9-22.9%, 6.3-14.2% and 8.1-22.7% respectively in theirs.

### Table 3: r-values and p-values of the bug patterns of Checkstyle with other seven projects

<table>
<thead>
<tr>
<th>Program Name</th>
<th>r-value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>0.450</td>
<td>0.019</td>
</tr>
<tr>
<td>Columba</td>
<td>0.615</td>
<td>0.001</td>
</tr>
<tr>
<td>Eclipse</td>
<td>0.489</td>
<td>0.010</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.524</td>
<td>0.005</td>
</tr>
<tr>
<td>Scarab</td>
<td>0.445</td>
<td>0.020</td>
</tr>
<tr>
<td>Luence</td>
<td>0.571</td>
<td>0.002</td>
</tr>
<tr>
<td>MegaMek</td>
<td>0.262</td>
<td>0.187</td>
</tr>
</tbody>
</table>

### Table 4: Corrected r-values for Pearson correlation of MegaMek in Pan et al.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>MegaMek</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>0.84</td>
</tr>
<tr>
<td>Columba</td>
<td>0.67</td>
</tr>
<tr>
<td>Eclipse</td>
<td>0.82</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.84</td>
</tr>
<tr>
<td>Scarab</td>
<td>0.80</td>
</tr>
<tr>
<td>Luence</td>
<td>0.77</td>
</tr>
<tr>
<td>MegaMek</td>
<td>1.00</td>
</tr>
</tbody>
</table>

5. ISSUES WITH THE CLASSIFICATION

During our manual classification, we had the opportunity to examine each fix hunk pair. If the hunk pair could be clas-
Table 2: Comparison of our classified data with Pan’s data

<table>
<thead>
<tr>
<th>Category Name</th>
<th>Sub-category Name</th>
<th>Our Result</th>
<th>Pan’s Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If related (IF)</strong></td>
<td>Addition of an Else branch (IF-ABR)</td>
<td>0.9%</td>
<td>0.6-1.8%</td>
</tr>
<tr>
<td></td>
<td>Addition of precondition check (IF-APC)</td>
<td>3.2%</td>
<td>2.2-6.0%</td>
</tr>
<tr>
<td></td>
<td>Addition of precondition check with jump (IF-APCJ)</td>
<td>3.2%</td>
<td>1.5-3.8%</td>
</tr>
<tr>
<td></td>
<td>Addition of post-condition check (IF-APTC)</td>
<td>0.5%</td>
<td>0.5-3.8%</td>
</tr>
<tr>
<td></td>
<td>Change of if condition expression (IF-CC)</td>
<td>10.0%</td>
<td>5.6-18.6%</td>
</tr>
<tr>
<td></td>
<td>Removal of an else branch (IF-RBR)</td>
<td>0.9%</td>
<td>0.3-1.1%</td>
</tr>
<tr>
<td></td>
<td>Removal of an if predicate (IF-RMV)</td>
<td>2.7%</td>
<td>1-2.9%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>21.5%</strong></td>
<td><strong>19.7 - 33.9 %</strong></td>
</tr>
<tr>
<td><strong>Method call (MC)</strong></td>
<td>Method call with different actual parameter values (MC-DAP)</td>
<td>5.9%</td>
<td>11.9-23.8%</td>
</tr>
<tr>
<td></td>
<td>Different Method call to a class instance (MC-DM)</td>
<td>1.8%</td>
<td>0.9-3.4%</td>
</tr>
<tr>
<td></td>
<td>Method call with different number or types of parameters (MC-DNP)</td>
<td>4.6%</td>
<td>2.3-7.2%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>12.4%</strong></td>
<td><strong>21.9 - 33.1%</strong></td>
</tr>
<tr>
<td><strong>Loop (LP)</strong></td>
<td>Change of loop condition (LP-CC)</td>
<td>1.4%</td>
<td>0.9-2.9%</td>
</tr>
<tr>
<td></td>
<td>Change of the expression that modifies the loop variable (LP-CE)</td>
<td>0%</td>
<td>0.1-0.3%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>1.4%</strong></td>
<td><strong>1 - 3 %</strong></td>
</tr>
<tr>
<td><strong>Assignment (AS)</strong></td>
<td>Addition/removal of switch branch (TY-ARCB)</td>
<td>0%</td>
<td>0.2-2.4%</td>
</tr>
<tr>
<td></td>
<td>Addition/removal of a try statement (TY-ARTC)</td>
<td>0.5%</td>
<td>1.1%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>0.5%</strong></td>
<td><strong>0.2 - 1.8 %</strong></td>
</tr>
<tr>
<td><strong>Switch (SW)</strong></td>
<td>Change of method declaration (MD-CHG)</td>
<td>16.4%</td>
<td>3.5-7.7%</td>
</tr>
<tr>
<td></td>
<td>Addition of method declaration (MD-ADD)</td>
<td>15.5%</td>
<td>3.9-11.6%</td>
</tr>
<tr>
<td></td>
<td>Removal of a method declaration (MD-RMV)</td>
<td>2.3%</td>
<td>0.7-5.4%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>34.2%</strong></td>
<td><strong>8.1 - 22.7 %</strong></td>
</tr>
<tr>
<td><strong>Sequence (SQ)</strong></td>
<td>Addition of operations in an operation sequence of field settings (SQ-AFO)</td>
<td>0%</td>
<td>0-9%</td>
</tr>
<tr>
<td></td>
<td>Addition of operation in an operation sequence of method calls to an object (SQ-AMO)</td>
<td>5.9%</td>
<td>2-0.7%</td>
</tr>
<tr>
<td></td>
<td>Addition or removal method call operations in a short construct body (SQ-AROB)</td>
<td>6.4%</td>
<td>0.9-4.2%</td>
</tr>
<tr>
<td></td>
<td>Removal of operations from an operation sequence of field settings (SQ-RFO)</td>
<td>0%</td>
<td>0-1.8%</td>
</tr>
<tr>
<td></td>
<td>Removal of operations from an operation sequence of method calls to an object (SQ-RMO)</td>
<td>0.5%</td>
<td>1.4-4.2%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>12.8%</strong></td>
<td><strong>5.4 - 19.1%</strong></td>
</tr>
<tr>
<td><strong>Class field (CF)</strong></td>
<td>Addition of a class field (CF-ADD)</td>
<td>9.6%</td>
<td>2.8-4.4%</td>
</tr>
<tr>
<td></td>
<td>Change of class field declaration (CF-CHG)</td>
<td>5.0%</td>
<td>1.9-4.7%</td>
</tr>
<tr>
<td></td>
<td>Removal of a class field (CF-RMV)</td>
<td>1.8%</td>
<td>0.8-3.4%</td>
</tr>
<tr>
<td></td>
<td><strong>Category Total</strong></td>
<td><strong>16.4%</strong></td>
<td><strong>6.5 - 11.4 %</strong></td>
</tr>
</tbody>
</table>

sified, we looked to see whether there were any ambiguities or other anomalies in the classification. Unclassified hunk pairs were examined to see whether there were any hunks could be classified into multiple patterns or into additional pattern/s, which could improve the proportion of fix hunks usefully classified. While, generally, the majority of hunks were classified in a straightforward manner, we did note a key class of anomalies, and have noted several patterns of fix hunk pairs which do not appear in Pan et al.’s [18] classification scheme. We present these, including an example, below. We present the examples in Unix diff format. A ‘+’ at the beginning of a line of code indicates that a line has been added in the bug fix, whereas a ‘-’ at the beginning of the line indicates that the line was removed as part of the bug fix change. Lines with neither symbol at their start were unchanged.

5.1 Multiply-categorizable bugs

A number of fix hunks contained changes that could be categorized in multiple ways by Pan et al.’s [18] scheme. Figure 2 shows a hunk pair where the fix involves adding a method call (which is classifiable as IF-CC). Pan et al. provided no guidance as to how such a change hunk should be categorized. Of the 419 effective fix hunk pairs we examined, we found 17 instances of this representing almost 4% of fix hunk pairs examined. The specific classifications involved differed from hunk pair to hunk pair.

```java
+ final int clsNameLen = aClassName.length();
+ final int pkgNameLen = mPkgName.length();
    final Iterator illIter = illegalInsts.iterator();
    while (illIter.hasNext()) {
        final String illegal = (String) illIter.next();
        + if (((illegal.length() - javaLang.length())
             == clsNameLen)
             && illegal.endsWith(aClassName)
             && illegal.startsWith(javaLang))
```

Figure 2: Example of Multiply-categorisable bugs
5.2 Possible additional pattern: method return value changes

A number of fix hunk pairs consisted of changes to the value related from a method, as shown in Figure 3. We found 11 such fix hunk pairs, representing almost 2.6% of hunk pairs examined. Such fix hunk pattern has not been reported or suggested under Pan’s scheme in [18]. It should be noted that in Pan et al.’s classification scheme, we found that the Switch related category has a range of 0 - 1.6% of the fix hunks, the Try/catch related category has 0.2 - 1.8%, the Loop related category has 1 - 3%. As such, Method return value changes can be a possible additional fix hunk pattern.

```java
public String getSourceName()
{
-    return mSourceName;
+    return mSourceClass.getName();
}
```

Figure 3: Example of method return value changes

5.3 Possible additional patterns: scope changes

If a sequence of statements is moved from one scope to another - for instance, moved into or out of a try block, or an if statement, these changes are not currently classified. In Figure 4, some of statements are moved from one try block to another try block. However, these kind of code changes, in our view, represent a natural fix “pattern”. While we found only 2 fix hunks that have this problem among 419 effective fix hunk pairs, we conducted an ad-hoc check of some of the remaining fix hunk pairs in the repository and found additional instances of such hunk pairs.

```java
try {
    fireFileStarted(aFileName);
    final String[] lines = getLines(aFileName);
    +    final Reader sar = new StringArrayReader(lines);
    +    VerifierSingleton.getInstance().clearMessages();
    +    VerifierSingleton.getInstance().setLines(lines);
    try {
        -    VerifierSingleton.getInstance().clearMessages();
        -    VerifierSingleton.getInstance().setLines(lines);
        -    final Reader sar = new StringArrayReader(lines);
        final GeneratedJava14Lexer jl
        = new GeneratedJava14Lexer(sar);
        jl.setFilename(aFileName);
    }
}
```

Figure 4: Example of scope changes

5.4 Possible additional patterns: string literals

Pan et al. [18] excluded all string literal changes, considering them as trivial. In our sample, we found that 9% of fix revisions involved string literal changes. However, it may still be useful to classify these explicitly. In many systems, they will indeed be trivial; as such, classifying them and removing them from the large mass of unclassified bugs will allow them to be excluded from any further consideration. However, there may well be times when such bugs are not trivial - consider, for instance, a mistake in a string literal in code that generates XML.

6. THREATS TO VALIDITY

6.1 Internal validity

We have manually analyzed only 373 bug fix revisions among 1631 bug fix revisions available in Checkstyle’s CVS repository. We believe that the subset we analyzed was large enough to get an approximate sense of the frequency of the most common bug fix patterns; however, it is not sufficient to rank the less common ones which did not occur, or occurred only a few times, in our sample. Furthermore, the post-2007 bug fixes in the Subversion repository may have revealed different patterns to be either more or less common, though there is no reason to expect there to be major differences.

Another possible threat to internal validity is the identification of fix hunk pairs. While we used the same method to identify bug fix hunks as Pan et al. [18], it is possible that the use of such keyword in the CVS change log was significantly different in Checkstyle to the software artifacts they analyzed. However, the similar levels of commit, file and hunk coverage is consistent with broadly similar handling of bug fix commits between ours and theirs. As such, we believe this potential threat is unlikely to affect our conclusions.

Finally, it is possible that our interpretation of Pan et al.’s [18] bug fix pattern may be inconsistent with their own; as such, we may have systematically mischaracterised some types of fix hunk pairs differently to how their tool would have if applied to Checkstyle. In fact, this probably occurred, given our identification of fix hunk pairs that could be classified using multiple patterns, as described in Section 5.1. However, we think this is unlikely to substantially affect our results relating to the similarity of the frequencies observed, given that the affected hunks represent only about 4% of the total hunks examined.

It is also possible that, due to the manual nature of our classification, some essentially “random” misclassifications may have occurred. We believe that this is unlikely to have substantially affected the conclusions which we have drawn.

6.2 External validity

This study only examined a relatively small number of fix hunk pairs in a single software artifact. Obviously, this limits the extent to which we can draw conclusions about Java software as a whole.

7. DISCUSSION

7.1 Soundness of classification scheme

Our results show that we have successfully applied Pan et al.’s [18] fault classification scheme to a new software artifact, Checkstyle. We were able to achieve similar levels of commit, file, and hunk coverage, and, of those that were classified, the vast majority were straightforward to categorize according to their scheme. Therefore, in response to our first research question, we believe that their classification scheme was for the most part sound, and was straightforward to replicate. The only issue with regards to the soundness of the classification was that a small but significant proportion of faults fell into multiple categories; no clarification was provided as to how their automatic classification tool proceeds in that situation, let alone a justification for any particular approach. Given that we found only 4% of faults
in our sample were affected by this problem, it seems unlikely that this would have substantially affected the results of Pan et al. However, in any future applications the scheme should be amended to remove this ambiguity.

There are at least two obvious approaches for reporting results in such a situation. Firstly, there is no particular reason why the fault categories must be mutually exclusive; as such, it is entirely possible to simply record a fault like that discussed in Section 5.1 as belonging to both categories. However, for some statistical analysis, double-counting hunk pairs may be undesirable. A second possibility would be to prioritize the fault categories, so that if a fix hunk could be categorized in multiple ways, the highest-priority category is the one recorded. While this removes the issue of double-counting, there is no immediately obvious, non-arbitrary way that such prioritization could be performed. Either way, future attempts to apply the categories should resolve this issue explicitly rather than leaving what must be an implicit prioritization process (and possibly even a nondeterministic one) based on the internals of their classification tool to decide.

7.2 Program-independence of fault category frequency

Our work did not support the hypothesis that the proportion of faults falling into the various categories would be independent of the type of software being categorized. As noted in Section 4.1, while the low Pearson r-values Pan et al. [18] reported the artifact “MegaMek” were the result of a calculation error, the difference between the frequencies found for Checkstyle and Pan et al.’s seven artifacts are much more striking than the originally-reported (and erroneous) differences seen with MegaMek.

Pan et al. [18] made the following comments based on their analysis of the original seven programs:

The strong frequency similarity suggests many further questions. The first concerns the validity of the result. Is MegaMek truly an outlier, or is the apparently similarity an artifact of selecting a set of systems that just happen to be very similar. Though the current data set of seven observed projects is large compared to prior bug categorization efforts, it is still not large enough to substantively answer this question.

Our results have cleared up one question - MegaMek is no longer an outlier - but have added another; and, indeed, the anomaly with Checkstyle is so large as to greatly limit the predictive value of the bug categorizations, were the variability shown with Checkstyle to be typical. Clearly, a significantly larger and more varied sample of software artifacts is required to definitively answer this important question.

Even so, there were some elements of quite strong similarity between our results and Pan et al.’s [18]. For instance, the category IF-CC (change of if condition expression) was extremely common in both our results and Pan et al.’s, representing 10% of faults in Checkstyle and between 5.6 and 18.6% in Pan et al.’s seven artifacts. It would seem reasonable to treat this result - which is consistent with the anecdotal experience of the authors as programmers - as quite robust and independent of the particular software analyzed.

7.3 Additional proposed bug patterns

Aside from the issue of ambiguous patterns discussed earlier, in Section 5 we have noted several patterns, similar in style to Pan et al. [18], which would allow the identification of a larger fraction of fix hunk pairs. The most straightforward addition would be a pattern for return value changes, which in the case of Checkstyle would identify an additional 2.6% of hunk pairs. A pattern or patterns for changes of scope, or the addition or removal of loops, have intuitive appeal but were rare in our original sample; however, as such changes usually represent substantial code changes it may well be worth adding these patterns.

9% of bug fix revisions had at least one hunk pair with a modified string literal. As such, we think the addition of this pattern is worth strong consideration, despite Pan et al.’s view that such changes are trivial and thus not worth classifying. If they are indeed trivial, being able to automatically classify them as such (and therefore not requiring further consideration) is still often useful information, and as discussed in Section 5.3, in some software string literal changes may often represent a real, non-trivial bug.

7.4 Using the classifications to improve testing

Even with the limitations and unresolved issues in the data available to date, we believe that the collated results of Pan et al., and our own study, have implications for testing practice, particularly for white-box testing. Testers could potentially benefit by choosing testing methods which are effective at detecting the most commonly-occurring faults, which the results indicate are faults relating to if conditions and method call parameter changes.

Perhaps the most robust result is the prevalence of faults relating to if conditions. Changes to if condition expressions alone represented 10% of all fixes to Checkstyle, and between 5.6% and 18.6% of fixes to Pan et al.’s artifacts. This supports the common intuition that if condition expressions should be rigorously tested. One potential technique for testing if condition expressions is the use of modified condition/decision coverage [4], which can select a reliable test set for boolean expressions such as that found in if conditions. Such a test set can be reliable on some “pre-defined” fault classes of boolean expressions. Therefore, it is worthy to devise an orthogonal bug fix patterns to analyze IF-CC fix patterns, where the fix patterns are mostly coincided with “pre-defined” fault classes. The preinformed fault class and its frequency information will guide testers to choose suitable testing methods of testing if conditionals.

Pan et al. [18] also noted the significance of this fault category, and conducted a further examination of these faults. They devised six subcategories of if condition expression faults, and collected frequency data for these subcategories on their seven artifacts. However, there is no obvious way to use these results to select specific testing methods. They found that fixes to if conditions resulted in a trend of increased complexity of conditionals. Further work is required to specifically investigate these faults in the context of condition coverage testing.

Another common bug fix type - though more common in Pan et al.’s data than our own - were changes in method call parameters. It is not immediately obvious how to test for bugs that are fixed in this manner. However, it would be interesting to see if method call parameter changes were related to changes in the called methods themselves, or were
the result of mistaken usage of methods that remain unchanged. It may also be useful to investigate some of the properties of methods whose invocations were modified, and the context in which the methods were invoked. These sort of analysis may help to identify the cause of these bugs. For instance, method overloading is a popular feature in the object oriented programming. When a method is overloaded, it means there is more than one declared method with the same name but different(type or number) parameters. It seems plausible that invocations of such methods are more error-prone, for instance because type checking may catch fewer cases where the method is mistakenly invoked with the wrong parameter. It is also important to investigate whether the method calls being changed relate to “internal” methods, third-party libraries, or even the standard Java class libraries; we intend to look into this question in future.

So, while there are already some useful observations to be made for testers and testing researchers in this data, there is also great scope for more detailed analysis of certain fault subsets to improve testing of specific fault types.

8. CONCLUSION

In this paper, we have investigated the fault pattern categories of Pan et al. [18] to determine their soundness and usefulness for testing research. We conducted a manual examination of bug fixes in Checkstyle, an open source Java software system. We found that the bug fix patterns were essentially sound; however, we also found that the proportion of bug fixes matching various patterns in Checkstyle were significantly different to any of the seven artifacts examined by Pan et al.. We noted that a small proportion of bug fixes matched multiple patterns, a situation not discussed by Pan et al., and proposed two alternative methods to resolve this ambiguity. We have also noted several potential additional patterns which should allow the classification of a substantially greater fraction of bug fixes.

Despite the limited data collected to date, we have identified that the bug fix patterns offer considerable potential for improving testing techniques; however, more detailed examination of certain common fix patterns is required to determine how best to improve testing in those cases. We propose two such areas for study: if conditions and method calls.

Overall, this preliminary study suggests that empirical analysis of bug fix patterns offers great potential for improving the standard of testing, particularly unit testing. As such, we believe such empirical analysis is worthy of considerable further research effort.

9. REFERENCES