Trends in Java code changes: the key to identification of refactorings?

Steve Counsell, Youssef Hassoun, Roger Johnson and Keith Mannock
School of Computer Science
Birkbeck College
University of London
Malet Street
London, WC1E 7HX.
email: steve@dcs.bbk.ac.uk

Emilia Mendes
Department of Computer Science
University of Auckland
New Zealand
email: emilia@cs.auckland.ac.nz

ABSTRACT
Changes made to object-oriented (OO) systems over time provide an insight into both design robustness and changes in requirements. When expressed at a high level of abstraction, observing trends in changes to code can indicate opportunities for refactoring at the architectural level. In this paper, we empirically investigate the changes made to a set of fifty-two Java library classes over a three year period. The research attempts to support the hypothesis that certain types of changes made to Java code fall into distinct trends and, furthermore, are likely to be made at a high level of abstraction; in this case to method signatures. Our empirical results show that change trends are identifiable thus informing well-known refactorings, but not as we had envisaged. Control logic constructs were found to be the focus of most changes to the library classes examined.

Categories and Subject Descriptors
D.2 [Software]: Software Engineering

General Terms
Languages, Measurement

Keywords
Java, Code, Changes, Refactoring

1. INTRODUCTION
The importance of identifying change features to OO systems is clear. Yet relatively little empirical or even anecdotal evidence exists to help categorise these changes [2, 5]. If we can identify commonly made changes to software, then this identifies opportunities for refactoring and even potential for meta-programming through dynamic modification of the behaviour of a class' methods.

In this paper, we describe the details of an empirical investigation using fifty-two Java library classes as a basis. The changes made to those classes over a period of three years were analysed and then categorised. A single hypothesis was then investigated, namely: for the library classes studied, the types of changes made fall into several broad categories; those made at the method signature level are the most frequent. Empirical data from our study supported this intuition, but only to a limited extent. Frequent changes were found to be made at the method signature level. However, nearly as many changes were found to be to method calls in the classes themselves and the most popular change was found to be to control logic in the classes and, in particular, to if statements.

In the following section, we outline our motivation and some related work. Details of the empirical investigation are then given including details of the data collected and analysis of the data collected (Section 3). Finally, in Section 4 we draw some conclusions and point to further work.

2. MOTIVATION AND RELATED WORK
The term software refactoring refers to a technique which seeks to improve code quality through the process of making one change, or a series of changes to the internal structure of the software without changing its external behaviour.

The key motivation for the work in this paper is the belief that changes to Java classes tend to be very similar in their nature and that studying trends in changes will highlight opportunities and potential for refactoring. At the very least, identification of these trends should allow us to draw conclusions about the feasibility of many refactorings in a practical sense (as opposed to what the theory says we should be attempting to refactor).

In terms of seminal refactoring literature, the work of Opdyke [8] describes a number of refactorings which should be applied to software. It also provided a demonstration of the potential for refactoring on a broad and semi-formal basis. In Najjar et al. [7], the potential benefits and problems from refactoring classes with a large number of constructors was investigated. Five Java systems were empirically investigated with respect to a refactoring, i.e., replacing multiple constructors with creation methods [6]. Results
from the empirical case study showed that removed (duplicated) lines of code across systems gave potential benefits in terms of improved class comprehension. However, the findings were not all supportive of the refactoring process. In some cases, refactoring of constructors was found to be prohibitively time-consuming and burdensome.

3. EMPIRICAL INVESTIGATION

The research contained in this paper is an empirical investigation of the association between proposed software features and other software indicators; in this case the potential for refactoring. We have used a quantitative analysis to support this investigation in keeping with other empirical studies such as found in [1].

3.1 Data Collected

The change data collected for this study was obtained manually from on-line source document representations of the diffs between version one of the fifty-two classes and the latest version of the same classes approximately three years later. The classes were taken from the gnu gcc libjava web pages; in particular, from the io library therein. For clarity, we looked at the original version of each Java class and the most recent version of the same class. We note that some of the classes analysed had evolved to different version numbers over that period depending on the number of changes made to the class; equally, some classes had undergone no changes during the same period.

The source documents from which the change data was extracted contained the changes made at a specific lines in the class body and, in addition, the differences between the versions in terms of how that line (or lines) had been changed. For simplicity, we sub-divided changes into three distinct types: additions to the class, e.g., a new method, deletions from the class and modifications to the class code, e.g., method signature modified. In this study, we were interested in all three types, since refactorings usually require a combination of code insertions and modifications and to a lesser extent deletions.

3.2 Summary Data

Table 1 gives summary data for the fifty-two classes including statistics of the breakdown into class additions (Add.), deletions (Del.) and modifications (Mod.). Table 1 shows the total number of changes to be three hundred and seventy-six. The maximum number of changes amongst all classes was sixty-nine. This belonged to a class called:

ObjectInputStream

a class used to read serialised objects. A particularly noticeable feature about this class was the number of if constructs added or changed over the period analysed (eleven), suggesting that a class based on verification of inputs may by its nature require extra checking capabilities to be added as it evolves. Since no new methods had been added over the period studied, this suggests that such a class needs to make more frequent use of its existing methods through method calls as it evolves. We remark from Table 1 that the number of additions to classes is comparable in size to the number of modifications made to classes. The number of deletions is relatively small, and from an evolutionary point of view, we would expect this to be the case - classes will tend to grow in size (in terms of number of methods) as they evolve and hence undergo relatively fewer deletions.

Table 2 shows the frequency of number of changes across the fifty-two classes. For clarity, we have arranged the frequencies into ranges. For example, thirty-four classes had between zero and four changes applied to them over the three year period; six classes had between twenty and sixty-nine changes applied to them in the same period, and so on. We note that of the fifty-two classes analysed, eight had undergone no changes over the period studied (this is included in the set of fifty-two classes). Inspection of these eight classes and some of the classes with one change only, revealed them to be classes whose functionality was quite limited in terms of number of methods and whose tasks were quite specific in nature - we might therefore have expected them to remain fairly static in terms of changes applied to them over this period. For example, four of the classes with zero changes applied to them were:

ObjectStreamField, ObjectStreamConstants, Serializable, SerializablePermission.

The nature of these classes lends support to this hypothesis.

3.3 Refactoring

A chief motivation for the research in this paper is to identify whether changes can be used to indicate the potential for key refactorings in Java code; a further motivation is to assess whether empirically, refactoring theory is reflected in practice; in other words, does refactoring actually happen? If not, then could it?

To investigate, a dependency diagram showing the relationships between the seventy-two refactorings outlined in Fowler et al. [4] was developed, and from this diagram emerged various core refactorings, i.e., refactorings which were central to the mechanics of other refactorings. The most relevant (to this paper) of these eight refactorings are listed below (we omit the other five through lack of space herein; they are listed in the Appendix). They are listed in an order reflecting the number of refactorings dependent on that particular refactoring. So, for example, the Extract Method refactoring is used in more refactorings than Move Field which in turn is used in more than Substitute Algorithm.

- 1. Extract Method. If a method is getting too long in terms of lines of code, split that method up to create two methods; this will rejuvenate the meaning of the method and hence retain a level of cohesion in the class.

- 2. Move Field. The motivation for moving a field from one class to another is when that field is being used by another class more than by the class in which it is defined.

- 7. Substitute Algorithm. This refactoring is used to replace an algorithm with one that is clearer and easier to comprehend.

In the next section, we describe in more detail the types of changes found in the fifty-two classes.
3.4 Types of Changes

Table 3 shows firstly, for the broad categories of changes, those which relate to the category of new method added; for succinctness, only the major change types have been outlined. The maximum value in this table denotes the greatest number of changes of that type found for any single class. It shows that thirty-two new methods were added over the period to the classes studied. Interestingly, these thirty-two new methods were accounted for by only seven classes of the fifty-two. Thus, while we have no hard evidence, it would seem from both visual inspection and the lack of new methods across the majority of classes that the extract method refactoring has not been applied in anger, at least not visibly so. Table 3 also shows that forty-five new method calls were added to the classes over this period. The class with thirty-two of those additions was named ObjectInputStream, coincidentally the class with the most if statements added (see Section 3.2). Nine classes accounted for these forty-five additions. Interestingly, while not qualifying as a refactoring (Improving The Design of Existing Code) refactoring which does not embrace a modification of the parameter. This refactoring did not figure amongst the eight most popular refactorings.

Most revealing from the data collected was the large number of if statements either modified or added to the classes. The sixty-seven additions or modifications to if statements were attributable to fourteen classes. This suggests that the Substitute Algorithm refactoring is as applicable as any of the other refactorings, based on the data presented. We note that in addition to added or modified ifs, very little evidence of added or modified while or for loops could be found in the changes; we were also surprised by the lack of attributes (moved) added (only twenty-six) across the set of classes, accounted for by only nine classes; there was only limited empirical evidence for the move field refactoring.

4. CONCLUSIONS

In this paper, we have described an empirical investigation into the changes made to a set of fifty-two Java classes. The motivation for the work was to investigate whether changes to code over a long period exhibited features of, or provided opportunities for refactoring. One threat to the validity of the work in this paper is that we have looked at only one system. We have also looked at a library-based system taken from a publicly available website. This could be criticised, since we might expect a class library to be less susceptible to change than any other type of application. In the case of a library system, the benefits of fine tuning are going to be accrued by a wide range of users (of those library classes). Furthermore, in other empirical studies, the features of library-based systems have been successfully compared with features of other system types (e.g., compilers, frameworks, graph editors) [3].

The main findings from the research were that although we suspected, a significant number of changes are made at the method signature level, nearly as many additions were found at the method call level. The highest number of changes, however, was attributable to additions or modifications of if statements. We therefore conclude that conscious refactoring is generally not being done by developers, but at least one common refactoring according to Fowler et al. is evident, namely, as part of the substitute algorithm refactoring. The conclusion is that developers may be refactoring without realising so. Future work will focus on extending this type of analysis to other application domains.

5. REFERENCES


<table>
<thead>
<tr>
<th>Total No. Changes</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Add.</th>
<th>Del.</th>
<th>Mod.</th>
</tr>
</thead>
<tbody>
<tr>
<td>376</td>
<td>0</td>
<td>69</td>
<td>7.23</td>
<td>158</td>
<td>19</td>
<td>199</td>
</tr>
</tbody>
</table>

Table 1: Summary data for system studied.

<table>
<thead>
<tr>
<th>Range</th>
<th>No. of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 4</td>
<td>34</td>
</tr>
<tr>
<td>5 - 9</td>
<td>8</td>
</tr>
<tr>
<td>10 - 14</td>
<td>3</td>
</tr>
<tr>
<td>15 - 19</td>
<td>1</td>
</tr>
<tr>
<td>20 - 69</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: Frequency of changes for classes studied.
<table>
<thead>
<tr>
<th>Change Type</th>
<th>Total No.</th>
<th>Distribution</th>
<th>Max.</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>New method added</td>
<td>32</td>
<td>7</td>
<td>14</td>
<td>0.62</td>
</tr>
<tr>
<td>Method call added</td>
<td>45</td>
<td>9</td>
<td>32</td>
<td>0.85</td>
</tr>
<tr>
<td>Param. in method call added or modified</td>
<td>32</td>
<td>19</td>
<td>9</td>
<td>0.62</td>
</tr>
<tr>
<td>Method signature modified</td>
<td>51</td>
<td>26</td>
<td>14</td>
<td>0.98</td>
</tr>
<tr>
<td>If added or modified</td>
<td>67</td>
<td>14</td>
<td>11</td>
<td>1.26</td>
</tr>
</tbody>
</table>

Table 3: Summary data for system studied.


Appendix: the five other refactorings

The five missing refactorings from the list given in Section 3.3 are: (3) Move method, (4) Self encapsulate field, (5) Replace constructor with factory method, (6) Rename method and (8) Replace conditional with polymorphism.