Do developers really worry about refactoring re-test? An empirical study of open-source systems

S. Counsell, S. Swift, M. Arzoky
Department of Computer Science
Brunel University, London
steve.counsell@brunel.ac.uk

G. Destefanis
Department of Computer Science
University of Hertfordshire, Herts, UK
g.destefanis@herts.ac.uk

Abstract. In the past two decades, refactoring has become a mainstream developer practice. One condition of doing refactoring is that the relevant code being refactored has to be re-tested afterwards to ensure that the code's semantics have been preserved. In this paper, we explore the extent to which a set of over 12000 refactorings fell into one of four re-test categories defined by van Deursen and Moonen; the 'least disruptive' of the four categories contains refactorings requiring only minimal re-test. The 'most disruptive' category of refactorings on the other hand requires significant re-test effort. We used multiple versions of three open-source systems to answer one research question: Do developers prefer to undertake refactorings in the least disruptive categories or in the most disruptive? The simple answer is, interestingly, that they prefer to do both. As well as providing insights into these refactoring patterns across the three systems, we also highlight a fundamental weakness with software metrics that try to capture the refactoring process.

Keywords: Refactoring, test, taxonomy, metrics, open-source.

1. Introduction

Since Fowler's seminal text on refactoring [3] and earlier work by Opdyke [7], the field of refactoring has, even conservatively speaking, spawned hundreds of studies [2, 4]. One facet of refactoring we know little about empirically, however, is the re-test implications of refactoring. Re-testing after refactoring is a necessary, yet time-consuming and potentially error-prone process and is heavily dependent on the type of refactoring being performed. One question that could inform our understanding of developer productivity, code quality and developer habits and which motivates this research is whether developers opt to undertake refactorings with a high re-test burden, *vis-à-vis* those that have only limited re-test requirements. An earlier paper by van Deursen and Moonen (vD&M) [9] explored Fowler's seventy-two refactorings and attached a test severity category to each. Their work was motivated by the fact that a refactoring should: "*not change its* [the code's] *observable behaviour. Ideally, this is verified by ensuring that all the tests pass before and after a refactoring. In practice, it turns out that such verification is not always possible: some refactorings restructure the code in such a way that tests can only pass after the refactoring if they are modified*". We used refactoring data extracted in a previous study by Bavota et al.,

[1] to carry out our analysis. The data was drawn from multiple versions of three open-source systems and is made available as a free download from [10]; it comprises 12046 refactorings extracted using Ref-Finder, a tool capable of extracting fifty-four of Fowler's seventy-two [8]. The same work by Bavota et al., investigated whether refactorings had been applied to code for which quality metrics (e.g., for size, coupling and cohesion) indicated the need for refactoring. A key result was that the metrics did *not* show a clear and obvious relationship with refactoring, suggesting that they cannot be used to identify classes that might need refactoring. Table 1 (taken from [1]) summarizes the three systems used, the versions analyzed and the ranges in classes and KLOC for each system. We note that in the original paper [1], a refactoring *and* code smell analysis was also undertaken; for the purpose of our study we used just the refactoring data giving rise to the 12046 refactorings.

System	Period	Releases	Classes	KLOC
Apache	Jan 2000-Dec 2010	18	87-1191	8-255
Xerces	Oct 2002-Dec 2011	23	777-1519	362-918
ArgoUml	Nov 1999-Nov 2010	11	181-776	56-179

Table 1. System summary (taken from [1])

2. vD&M's test taxonomy

In their paper, vD&M [9] describe four separate categories into which the seventy-two refactorings of Fowler can be placed. Initially, five categories (A-E) were described in their paper. However, Category A refactorings were dropped from their analysis on the basis that they were simply an amalgam of smaller refactorings. The four remaining categories (B-E) are defined in increasing levels of re-test burden as:

1. Compatible (Category B refactorings): do not change the original interface.
2. Backwards Compatible (Category C refactorings): change the original interface and are inherently backwards compatible, since they extend the interface.
3. Make backwards compatible (Category D refactorings): change the original interface and can be made backwards compatible by adapting the old interface. For example, the 'Move method' refactoring that moves a method from one class to another can be made backwards compatible through the addition of a 'wrapper' method to retain the old interface.
4. Incompatible (Category E refactorings): change the original interface and are not backwards compatible because they may, for example, change the types of classes involved making it difficult to wrap the changes (e.g., Move field).

So, in theory Category B refactorings should present less of a re-test burden than those in Category C and those in Category C less than in Category D etc. Table 2 shows the seventy-two refactorings of Fowler when placed into each of the four categories (B-E) as detailed by vD&M.

Category/Set of refactorings
Category B: Change Bi-directional Association to Unidirectional, Replace Magic Number with Symbolic Constant, Replace Nested Conditional with Guard Clauses, Consolidate Duplicate Conditional Fragments, Replace Conditional with Polymorphism, Replace Delegation with Inheritance, Replace Inheritance with Delegation, Replace Method with Method Object, Remove Assignments to Parameters, Replace Data Value with Object, Introduce Explaining Variable, Replace Exception with Test, Change Reference to Value, Split Temporary Variable, Decompose Conditional, Introduce Null Object, Preserve Whole Object, Remove Control Flag, Substitute Algorithm, Introduce Assertion, Extract Class, Inline Temp.
Category C: Consolidate Conditional Expression, Replace Delegation with Inheritance, Replace Inheritance with Delegation, Replace Record with Data Class, Introduce Foreign Method, Pull Up Constructor Body, Replace Temp with Query, Duplicate Observed Data, Self Encapsulate Field, Form Template Method, Extract Superclass, Extract Interface, Push Down Method, Push Down Field, Extract Method, Pull Up Method, Pull up Field.
Category D: Change Unidirectional Association to Bi-directional, Replace Parameter with Explicit Methods, Replace Parameter with Method, Separate Query from Modifier, Introduce Parameter Object, Parameterize Method, Remove Middle Man, Remove Parameter, Rename Method, Add Parameter, Move Method.
Category E: Replace Constructor with Factory Method, Replace Type Code with State/Strategy, Replace Type Code with Subclasses, Replace Error Code with Exception, Replace Subclass with Fields, Replace Type Code with Class, Change Value to Reference, Introduce Local Extension, Replace Array with Object, Encapsulate Collection, Remove Setting Method, Encapsulate Downcast, Collapse Hierarchy, Encapsulate Field, Extract Subclass, Hide Delegate, Inline Method, Inline Class, Hide Method, Move Field.

Table 2. The four vD&M categories and refactorings in each [9]

2.1 Category analysis

We begin our analysis by detailing the number of refactorings found in each category according to the data of Bavota et al. Table 3 shows, for each of the three open-source systems, 1) the number of refactorings applied across the four categories (B, C, D and E), 2) the percentages that this represents and, 3) the totals for each category and each system. For example, in Apache, 673 refactorings were applied from Category B. This represents 52.21% of the total of 1289 refactorings undertaken in the entire system. Equally, 17.41% is the corresponding proportion of 3865 Category B refactorings that 673 represents.

System	Category B		Category C		Category D		Category E		Total
Apache	673		105		423		88		1289
	52.21	17.41	8.15	13.01	32.82	7.57	6.82	10.90	
Xerces	2056		499		3663		1284		7502
	27.41	53.20	6.65	61.83	48.83	**65.52**	17.11	**72.01**	
ArgoUml	1136		203		1505		411		3255
	34.90	29.39	6.24	25.15	46.24	26.92	12.63	23.05	
Total	3865 (32.09)		807 (6.70)		5591 (46.41)		1783 (14.80)		**12046**

Table 3. Number of refactorings in each category (all systems)

For Apache, Category D was only the second highest in terms of refactorings (exceeded by the number in Category B). Table 3 also shows that the highest number of refactorings for Xerces and ArgoUml was found in Category D (3663 and 1505 refactorings, respectively). This category accounted for 46.41% of the total number of refactorings across the three systems. For Xerces, 48.83% of all refactorings were in Category D and 65.52% of Category D refactorings were attributable to the same system (value bolded in the table). The lowest number of refactorings in all systems was for Category C, which accounted for just 807 (6.70%) of the total 12046. While Category B accounted for a significant proportion of the total (32.09%), it was Category D that seems to dominate the overall set. At the other extreme, Category E accounted for just 1783 (14.80%) of total number of refactorings; finally, Xerces accounted for 72.01% of all Category E refactorings (value bolded in the table).

2.1.1 Result summary

From the data presented, it is evident that developers did undertake many low test impact refactorings. Category B accounts for nearly a third of all refactorings. However, nearly 50% of the total number of refactorings across all systems were drawn from Category D. This propensity for Category D refactorings was a surprising and revealing result and contrary to our intuition. We might have expected developers to prefer to undertake Category B and C refactorings because they are simpler in a re-test sense (in fact Category C actually saw the lowest number of refactorings). This does not seem to be the case, however from the data.

2.2 Refactoring analysis

One question which can then be asked is which refactorings were applied most frequently across the four categories? That might help us understand why the result of the previous section was found. For Apache, three refactorings stood out in Category D, namely: Add parameter, Remove parameter and Rename method. These three refactorings accounted for 30.72% of all refactorings applied in the system. The most frequent was Rename method, whose motivation is described by Fowler [3] as: "*The name of a method does not reveal its purpose*". The solution is simply to: "*Change the name of the method*". In Category B, most of the refactorings related to the low-level manipulation of conditional logic in the code. For example, 314 of the 673 refactorings were attributable to the: Replace magic number with symbolic constant (RMNwSC) refactoring. The motivation for this refactoring [3] is: "*You have a literal number with a particular meaning*". The solution is to: "*Create a constant, name it after the meaning, and replace the number with it*." The example given in [3] to illustrate is as follows:

```
double potentialEnergy(double mass, double height) {
  return mass * height * 9.81;
}
```

After the refactoring, this code becomes:

```
double potentialEnergy(double mass, double height) {
  return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

For Xerces, three refactorings stood out in Category D. These were Rename method (in keeping with Apache), Move method and Add parameter (again, the same as Apache) with 1061, 1183 and 929 refactorings, respectively. In addition, a significant number of Move field refactorings (Category E) were also found (1183). In terms of Category B refactorings, the RMNwSC refactoring again stood out with 597 refactorings. Another noticeable Category B refactoring was Consolidate conditional duplicate fragments (CDCF) with 474 instances. The motivation for CDCF according to Fowler [3] is: "*The same fragment of code is in all branches of a conditional expression*". The solution is to: "*Move it outside of the expression*". The following example illustrates this refactoring [3]:

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

After being refactored, the code without the duplicated method `send()` becomes:

```
if (isSpecialDeal())
  total = price * 0.95;
else
  total = price * 0.98;
send();
```

In ArgoUml, the same core set of refactorings seemed to arise. In Category D, the Add parameter, Remove parameter and Rename method refactorings again featured as those most applied with 491, 427 and 261 refactorings, respectively. Together, these three accounted for 1179 of the 1505 D category refactorings (i.e., 78.33%). For the B and C categories, only two refactorings stood out. The Replace method with method object refactoring (Category B) accounted for 367 refactorings. The purpose of this refactoring is to turn a method into its own object so that the local variables it uses become fields on that object. Equally, the Remove control flag (RCF) refactoring accounted for 224 of the total number of refactorings. The motivation for RCF is when "*You have a variable that is acting as a control flag for a series of boolean expressions*". The solution is to: "*Use a break or return instead"*. Finally, the RMNwSC refactoring again featured with 145.

2.2.1 Result summary

A small subset of refactorings therefore dominates the total set of refactorings across all three systems. In Category B, refactorings that manipulated low-level program

logic accounted for the majority e.g., Replace magic number with symbolic constant' and, correspondingly, in Category D, where Add Parameter, Remove parameter and Rename method accounted for the majority. This result confirms Bavota et al's conclusion with respect to metric applicability. Very few current, popular metrics seem to capture low-level code logic constructs (i.e., that of conditionals, nesting, flag manipulation). Many of these refactorings manipulate low-level code (e.g., RCF, RNCwGC and RMNwSC etc) and so it goes without saying that such metrics will be unlikely to provide insights into refactoring behaviour. Metrics that capture coupling, cohesion and size etc therefore largely miss the point of refactoring. It is no surprise that Bavota et al., found no relationship between metrics and refactoring.

2.3 Evolutionary analysis

One aspect of the data that might further inform our analysis is whether, over the course of time, the trend in application of refactorings changes. We therefore looked at whether developers tended to undertake less of the Category D and E refactorings and more in the B and C categories on an evolutionary basis. The premise of this analysis is that, as systems age, they become more difficult to maintain as they erode and developers will therefore undertake refactorings with less complexity and with less of a test burden than others. To answer this question, we ordered the set of refactorings according to the version they were applied in. Figures 1a, 1b and 1c show the distribution of the four categories across versions for the three systems. The *x*-axis is the version number (we have simply numbered these starting from 1) and the *y*-axis the number of refactorings in each of the four types.

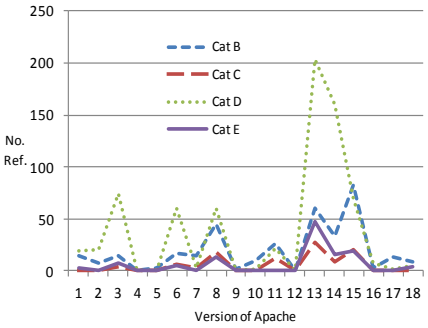

Figure 1a. Refactorings in Apache

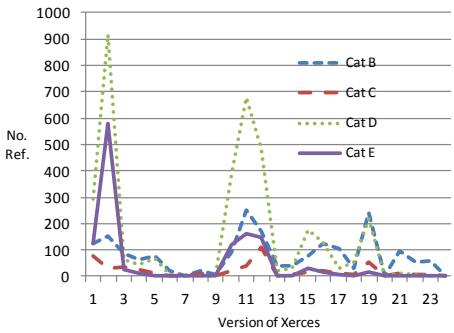

Figure 1b. Refactorings in Xerces

Figure 1a shows the data for Apache. Most pronounced from the figure are the peaks in Category D refactorings which occur throughout the course of the versions studied, but are particularly evident in versions 13, 14 and15. The same is true to a lesser extent for Category B with a number of peaks, particularly in later versions. Version 15 stands out with 80 refactorings in this sense. The other two categories remain relatively static in numbers apart from one peak for Category E in version 13 with 47 refactorings. However, for this system, there does not appear to any less inclination to undertake Category D refactorings as the system ages (Category E showed very few refactorings overall anyway). On the other hand, there does seem to be an increase in

the number of Category B refactorings as the Apache system evolves given by the relatively large peaks in version 13 onwards.

Figure 1b shows the same data for Xerces. Again, the presence of peaks in the first and middle versions for Category D is notable. Category E also shows extremes in version 2 and to a lesser extent 11. As for Apache, Category D is relatively erratic in nature with peaks and troughs throughout the versions studied. The same is true of Category B. For Category D, peaks in version 2 and 11 can be seen and the same for Category B in versions 11 and 19. The pattern of erratic refactorings for Xerces is similar to Apache. Again, however there does not appear to any less inclination to undertake Category D refactorings as the system ages. Finally, Figure 1c shows the data for ArgoUml. The peaks, particularly in Categories B, D and E are noticeable from the graph. For Categories B and D, there are large peaks in version 5 (a lesser peak for Category D is also evident in version 2). Category E also features some peaks in versions 2, 5 and 8. While the number of Category D refactorings in later versions is less pronounced, there is still no clear evidence that developers avoided relatively test intensive refactorings in later versions of the system.

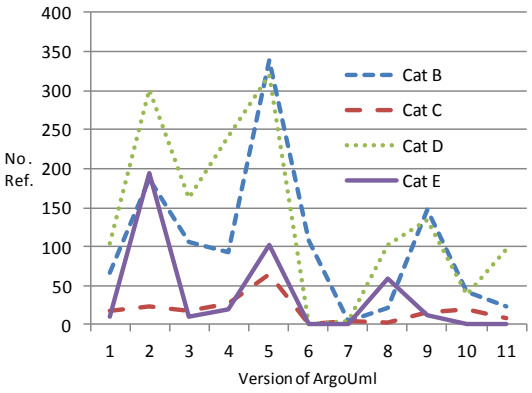

Figure 1c. Refactorings in ArgoUml

Across all three systems, there does not seem to be a reduction in Category D and E refactorings or a dramatic rise in Category B and C refactorings.

3. Conclusions and further work

In this paper, we explored the extent to which a set of over 12000 refactorings fell into one of four re-test categories previously defined by van Deursen and Moonen. We explored whether developers would prefer to carry out refactorings with a low test burden rather than those where significant re-test might be involved (Category B and C refactorings versus D and E). The analysis showed as a primary result that open-source developers seem to apply refactorings largely irrespective of the test category and hence the re-test burden. Clearly, developers do not really care about refactoring re-test or, if they do, this does not affect their choice of refactoring. No trends in that

direction were found on an evolutionary basis either. Of course, we have no information on whether developers used tools to assist in the refactoring process or whether they were manually performed. We have also only studied three open-source systems and limited versions of those systems. However, in defence of this threat, various other studies of developer habits suggest that developers generally prefer to refactor manually, rather than using tools. In one study by Murphy-Hill et al., [5] approximately 90% or all refactorings were applied manually. In another study by Negara et al., [6], experienced developers were found to apply 11% more manual refactorings than automatic, especially in renaming operations.

A secondary and more wide-ranging result of the research was that current metrics seem to capture OO class features well, but they are not at the right level for analysing refactoring; this was a key result of Bavota et al., [1] and on which our research is based. This effectively means that OO metrics are largely redundant for indicating the need for refactoring. One avenue of future work is to encourage fresh metric initiatives to establish those that do – and these should be targeted at conditional nested code constructs. In addition, it would be interesting to explore whether, using data mining techniques, certain refactorings were always applied together.

References

[1] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, F. Palomba, An Experimental Investigation on the Innate Relationship between Quality and Refactoring,. Journal of Systems and Software, 107, C (September 2015), 1-14.

[2] S. Demeyer, S. Ducasse and O. Nierstrasz, Finding refactorings via change metrics, ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Minneapolis, USA. pages 166-177, 2000.

[3] M. Fowler, Refactoring: improving the design of existing code, 1999.

[4] T. Mens and T. Tourwe, A Survey of Software Refactoring, IEEE Transactions on Software Engineering 30(2): 126--139 (2004).

[5] E. Murphy-Hill, C. Parnin, A. Black, How We Refactor, and How We Know It. IEEE Trans. Software Eng. 38(1): 5-18 (2012).

[6] S. Negara, N. Chen, M. Vakilian, R. Johnson, D. Dig, A Comparative Study of Manual and Automated Refactorings, ECOOP 2013.

[7] W. Opdyke. Refactoring object-oriented frameworks, Ph.D. Thesis, Univ. of Illinois. 1992.

[8] K. Prete, N. Rachatasumrit,N. Sudan, M. Kim, Template-based Reconstruction of Complex Refactorings, International Conference on Software Maintenance, Timisoara, Romania, pp. 1-10, 2010.

[9] A. van Deursen and L. Moonen. The Video Store Revisited - Thoughts on Refactoring and Testing. International Conf. on eXtreme Programming and Flexible Processes in Software Engineering XP 2002, Sardinia, Italy.

[10] https://figshare.com/articles/An_Experimental_Investigation_on_the_Innate_Relationship_between_Quality_and_Refactoring/1207916