Is Complexity of Re-test a Reason Why Some Refactorings Are Buggy? an Empirical Perspective

Steve Counsell(✉), Steve Swift, Mahir Arzoky, and Giuseppe Destefanis

Department of Computer Science, Brunel University, London, UK
steve.counsell@brunel.ac.uk

Abstract. In this short paper, we explore one simple, yet unexplored question about the relationship between refactoring and bugs. Is the complexity of re-testing code immediately after refactoring a reason why some refactorings are buggy? To facilitate our analysis, we use a set of over four thousand refactorings mined from three open-source systems and decomposed into the four test categories of van Deursen and Moonen. Preliminary results showed that, compared with non-buggy classes, buggy classes had been subjected to more refactorings where a large re-test commitment was required; extent of re-test may therefore be a significant factor in determining whether refactoring creates bugs. Our finding supports that of Bavota et al. - that more and better testing after certain refactoring practices could reduce the harm that refactorings cause.

1 Introduction

Since the 1990's and the seminal texts on refactoring were published by Opdyke [8] and Fowler et al. [5], refactoring has been the subject of hundreds of empirical studies and become a vital tool in the daily work of a developer. Refactoring can be defined as the process of: "*Changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [5]. Despite this multitude of prior studies, there are still a range unanswered research questions. One of these is the link between refactoring and bugs. So, does refactoring cause bugs and if it does, then which types of refactorings cause bugs the most often? Every refactoring requires the developer to re-test the affected code to ensure that program behaviour has been preserved. So, an equally relevant research question to ask is whether the extent of re-test required post-refactoring influences the level of bugs in a system. To assess the level of post-test necessary, we use a taxonomy developed by van Deursen and Moonen [9]. The taxonomy categorizes each of 72 refactorings according to how difficult it is to unit post-test and it thus indicates the level of effort required to ensure that the refactoring has been successfully applied. Results from our analysis showed a tendency for buggy classes to have a higher number of refactorings with extensive re-test (compared to non-buggy classes). This implies that

M. Shepperd et al. (Eds.): QUATIC 2020, CCIS 1266, pp. 83–90, 2020.
https://doi.org/10.1007/978-3-030-58793-2_7

if we do complex and lengthy refactorings, then we should take every step to ensure that program behaviour has been preserved. The remainder of the paper is organized as follows. In Sect. 2, we describe information on the systems studied and taxonomies/data. We then present results through an analysis of the three systems (Sect. 3), before discussing results in Sect. 4. Finally, we conclude and point to further work in Sect. 5.

2 Preliminaries

2.1 Taxonomy of van Deursen and Moonen

To assess the extent of unit, post-refactoring testing required for every refactoring, we use the taxonomy developed by van Deursen and Moonen (V&M) [9]. The purpose of the taxonomy is to allocate each of Fowler's 72 refactorings [5] to a category, depending on the complexity and extent of the post-test required after that refactoring had taken place. The taxonomy is motivated by V&M as follows: "*One of the dangers of refactoring is that a programmer unintentionally changes the systems' behavior. Ideally, it can be verified that this did not happen by checking that all the tests pass after refactoring. In practice however, there are refactorings that will invalidate tests (e.g., when a method is moved to another class and the test still expects it in the original class)*". In short, the categories represent an increasingly complex post-refactoring test effort commitment on the part of the developer and the taxonomy reflects the fact that some refactorings restructure the code in such a way that unit tests can only pass after the refactoring once those tests have been modified. The four categories identified by V&M are as follows:

1. **Compatible**: Refactorings that do not change the original interface. In the compatible refactoring category, we can use existing tests to check the refactoring. One example is the Extract method refactoring [5], which takes a section of code from a method and forms a new method (or methods) with that code. However, since this refactoring creates at least one new method, we need to add tests that document and verify that the new method has actually preserved behaviour.
2. **Backwards compatible**: Refactorings that change the original interface and are inherently backwards compatible since they extend the interface. In the case of this set of refactorings, according to V&M, "*... the tests keep running via the original interface and will pass if the refactoring preserves tested behavior. Depending on the refactoring, we might need to add more tests covering the extensions*". One example of this refactoring type is Consolidate Conditional Expression, which re-arranges a conditional statement into a simpler and less complex format. The conditions in the statement do not change and it *might* be possible to use the same test on the changed code.
3. **Make backwards compatible**: This applies to refactorings that change the original interface and can be made backwards compatible by adapting the old interface. For example, the Move method [5] refactoring physically

moves a method from one class to another class and can be made backwards compatible by adding a wrapper method to retain the old interface.

4. **Incompatible**: This applies to refactorings that change the original interface and are not backwards compatible because they may, for example, change the types of classes that are involved. This makes it difficult to wrap the changes. The refactoring destroys the original interface and, as such, all tests that rely on the old interface need to be adjusted.

In theory, Compatible refactorings require less post refactoring test effort than those in the Backwards compatible category, which in turn require less effort than refactoring in the Make backwards Compatible category etc. In line with this, we adopt the stance that Compatible refactorings are the most straightforward to re-test and Incompatible refactorings the most complex; this is essentially the message that V&M convey in their work.

2.2 Systems Analysed

Our analysis in this paper is facilitated by the earlier work of Bavota et al. [2]. In their analysis, they describe how some refactorings were more likely to induce a bug than others, using the set of Fowler's refactoring activities as the vehicle. Results indicated that, while some kinds of refactorings were not likely to be harmful, others tended to induce bugs frequently (i.e., were harmful). In their study, they used refactoring and bug data extracted from three open-source systems and made that data available. We use that same data in this paper to explore refactorings but from a purely re-test perspective. The systems studied were: Xerces, ApacheAnt and ArgoUML. Xerces is a Java XML parser, ApacheAnt a build tool and library primarily designed for Java applications and ArgoUML a UML modelling tool[1]. Table 1 shows summary statistics for the three systems including: the period of time over which the system was studied, the releases analysed, the number of releases and, finally, the number of classes. The summary statistics in Table 1 are reproduced from the paper by Bavota et al.

Table 1. Three systems studied (summary data)

System	Period	Analyzed	Rel.	No. classes
Xerces	Nov '99–Nov '10	1.0.4-2.9.1	33	181-776
ApacheAnt	Jan '00–Dec '10	1.2-1.8.2	17	87-1191
ArgoUML	Oct '02–Dec '11	0.12-0.34	13	777-1519

[1] http://ant.apache.org/, http://argouml.tigris.org/, http://xerces.apache.org/xerces-j/.

2.3 V&M Decomposition

Table 2 shows the number of refactorings across each system and the number of refactorings in each of V&M's categories; here, Comp. or comp denotes Compatible, b/comp. denotes Backwards compatible and Incomp. represents Incompatible. The refactorings were collected in the original study of Bavota et al. by the Ref-Finder tool [6] and we use exactly the same set of refactorings. In the Xerces system for example, there were 1528 refactorings in total; of that total, 668 were "Compatible", 218 "Backwards compatible", 510 "Make backwards compatible" and 132 "Incompatible".

Table 2. Categories of V&M across the three systems

System	Comp.	Backwards comp.	Make b/comp.	Incomp.	Total
Xer.	668	218	510	132	1528
Apac.	290	67	142	21	520
Argo.	788	194	837	194	2013
Total	**1746**	**479**	**1489**	**347**	**4061**

From Table 2, we also see that the Compatible category forms the largest number (1746) and that ArgoUML has the highest number of overall refactorings (2013) across the four categories. Significant numbers of Make backwards compatible refactorings can also be seen; Incompatible refactorings were relatively small in number (just 347 from 4061 (8.54%)). The numbers in the table thereby give a mixed picture in terms of a developer's propensity to choose refactorings with a less complex required post-refactoring test. It seems that they are as likely to undertake complex refactorings (with long re-test requirements) as they are simpler ones. The numbers of Incompatible refactorings are comparable with the numbers of Backwards compatible refactoring, even though the former represents the lengthiest post refactoring test required. Based on the evidence from Table 2, it appears that developers do not necessarily choose refactorings with a low post-refactoring test burden according to the taxonomy of V&M.

3 Data Analysis

3.1 Buggy Classes

To further our analysis, we looked at the type of refactorings in each system for classes where *at least* one bug had been recorded due to a refactoring and the V&M categories of those refactorings. Table 3 lists the name of the refactorings and, for each of the three systems abbreviated to Xer,. Apa., and Arg., the number of refactorings, total and the V&M category of that refactoring. In the Xerces system, there were 26 Add parameter refactorings made to classes with at least one bug and across all three systems there were 68 Add parameter refactorings in total. Add parameter falls into the "Make backwards compatible" category of refactoring. We note that for space in the paper (there were in excess of 35 types of refactoring), we have left out any refactoring where, across the three systems, there were less than ten refactorings of that type - we do include these in reported analyses, however. We have also abbreviated some refactorings in Table 3 for space purposes. In full and in the order of the table, these are: Consolidate conditional expression, Consolidate duplicate conditional fragments, Introduce explaining variable, Remove assignments to parameters, Replace method with method object, Replace nested conditional with guard clauses and Replace magic number with symbolic constant.

Table 3. V&M categories (buggy classes)

Refactoring	Xer.	Apa.	Arg.	Total	V& M category
Add parameter	26	10	32	68	Make b/comp.
Cons. cond. expression	11	5	3	19	Backwards comp.
Cons. dup. cond. frag.	16	11	13	40	Comp.
Extract method	11	11	19	41	Backwards comp.
Inline method	6	2	2	10	Incomp.
Intr. explaining variable	5	7	14	26	Comp.
Intr. null object	9	8	13	30	Comp.
Inline method	2	0	10	12	Comp.
Move field	11	3	63	77	Incomp.
Move method	7	8	40	55	Make b/comp.
Remove ass. parameters	8	4	9	21	Comp.
Remove control flag	3	5	11	19	Comp.
Remove parameter	24	7	31	62	Make b/comp.
Rename method	19	9	17	45	Make b/comp.
Rep. meth. w meth. o.	16	0	80	96	Comp.
Rep. nest. cond. w g.	12	0	8	20	Comp.
Rep. mag. no. w sym. c.	16	13	7	36	Comp

In terms of the totals and including 13 refactorings left out of the table for the Compatible category, there were 313 (43.84%) refactorings. For the Backwards compatible category and including 8 refactorings left out of the table, there were 68 refactorings in the category (9.52%). For the Make backwards compatible category and including 9 refactorings not in the table, there were 239 refactorings in that category (33.47%). Finally, for the Incompatible category, including 7 not in the table, there were 94 refactorings (13.17%). Overall, this means that 53.36% of all refactorings with at least one bug were drawn from the Compatible and Backwards compatible categories. Most notable from the data, however, is that 46.64% of refactorings in classes with at least one bug were in the Make backwards compatible and Incompatible categories (i.e., those requiring the most post-test effort).

3.2 Non-buggy Classes

Table 4 shows the corresponding data for classes where *no bug* was recorded. So, for the Xerces system, 581 refactorings had been applied to classes with no recorded bug in the Compatible category. Similarly, 115 refactorings in the Incompatible category had been applied to classes with no yet recorded bugs. It also shows the total in each category and the percent that those totals reflect of the total number of refactorings. For comparison, we also include the percent for the buggy refactorings from Table 3. For example, there were 1311 Compatible refactorings across the three systems and this represents 42.57% of the total. For the buggy total in Table reftbl3, the corresponding percent was 43.84.

Table 4. V&M categories (non-buggy classes)

System	Comp.	Backwards comp.	Make b/comp.	Incomp.
Xerces	581	196	434	115
ApacheAnt	242	51	108	16
ArgoUML	488	134	607	107
Total	**1311**	**381**	**1149**	**238**
% non-buggy	**42.58**	**12.37**	**37.32**	**7.73**
(% buggy)	**(43.84)**	**(9.52)**	**(33.47)**	**(13.17)**

The most notable feature of Table 4 is the contrast between the number of Make backwards compatible and Incompatible refactorings compared with the set of data from Table 3. From the set of non-buggy classes totalling 3079 refactorings, we see that the Compatible category accounted for 42.58% the Back compatible category accounted for 12.37%, the Make backwards compatible category 37.32% and the Incompatible category just 7.73%. In total therefore, 54.94% were drawn from the Compatible and Backwards compatible categories and 45.06% from the Make backwards compatible and Incompatible categories.

From this data, the percentages are similar across buggy and non-buggy classes for three of the four categories. It is the relatively larger number of Incompatible refactorings (13.17%) in buggy classes compared with non-buggy classes - the corresponding value in non-buggy was almost half. We posit that the cause of bugs in the buggy classes was due directly to the extra number of Incompatible buggy refactorings. In particular, we single out the Move field refactoring in the ArgoUML system with 66 individual instances of this refactoring. As its name suggests, this refactoring should be applied when: "*A method is* *used more in another class than in its own class*". The solution is to move that method to the class where it is being used most. Move method is a refactoring whose key purpose is to reduce coupling, a feature of systems that is widely acknowledged (when in excess) as contributing to the code buggyness [1,3]. So, while the Move method may well solve one problem, it may cause others due to the re-test required and the bug potential as a result.

We carried out Chi-square test to determine statistical significance of buggyness and its influence on the different categories [4]. We used a $2 * 4$ contingency table, with buggy and non-buggy representing the two rows in the table and the four columns representing the categories of refactoring and totals. The contingency table is therefore an amalgamation of the results found in Tables 3 and 4, representing buggy and non-buggy sets of refactorings and the numbers of refactorings in each, respectively. The Chi-square analysis gave a p-value of 0.00001 (degrees of freedom = 3). This value is less than <0.01 and we therefore fail to accept the null hypothesis of independence between buggyness and refactoring category. The buggyness of a class *is* dependent on the category of refactoring.

4 Discussion

Our short paper precludes a full treatment of the literature in the area. However, our analysis is heavily informed by the work of Bavota et al. [2]. They explored the buggyness of Fowler's set of 72 refactorings. Their conclusion was that more accurate code inspection and testing activities needed to be in place to prevent refactorings causing harm to code and seeding bugs. Our analysis has shown the same rule applies, but that it may be the extent of post-refactoring test that may be a contributory factor to bugs. We would qualify this by saying that if you undertake refactorings in the Incompatible category according to V&M, then extra care and attention should be invested in the testing process to make sure it is done properly. The refactoring literature on trends and traits in refactorings and the closely linked topic of code smells is well-documented [5]; however, the issue of the damage that post-refactoring can do is still largely undocumented. In this paper, we take the first steps in that research direction.

We also need to consider the threats to the validity of our study. Firstly, we have only examined three systems, which is a small sample. There is no guarantee that, were we to study more systems, the same results would be found. Secondly, the taxonomy of V&M studied in this paper is theoretical only and, unlike the study of Bavota et al. [2], is not empirically-based. This could be criticised since it is an untried taxonomy "in the field". Thirdly, we have studied only open-source code; industrial code may show altogether different features. Fourthly, the message that this work has conveyed is that refactorings with high post-test may cause bug-related problems. But there may be multiple other factors to consider in the development process. For example, the experience of the developer, the age of the system or the refactoring strategy adopted by the organization. Finally, we cannot be sure of the proportion of automated refactorings used in this study *vis-a-vis* those carried out manually; we assume a manual approach to refactoring. However, there is empirical evidence suggesting that a high proportion of developers prefer manual refactoring anyway [7].

5 Conclusions and Future Work

In this paper, we explored a single research question related to refactoring. The question asked whether the harm that refactoring can do was related to the amount of re-test necessary after applying a refactoring. The taxonomy of van Deursen and Moonen was used to support our analysis and this placed every refactoring into one of four categories in ascending difficulty of testing. Preliminary results showed that, compared with non-buggy classes, buggy classes had been subjected to more refactorings where a large re-test commitment was required. This implies that refactorings causing bugs may be simply down to the test load and human errors that may arise from that; for larger, more complex refactorings there is more room for human error than for smaller, less complex ones. Future work will focus on experiments with industrial developers to see if refactorings with low post requirements are, experimentally, more likely to induce a bug. This will take the form of a replication of Bavota's study. Finally, it would be interesting to extend our study to other open-source systems and to multiple application domains.

References

1. Basili, V., Briand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Soft. Eng. **22**(10), 751–761 (1996)
2. Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O.: When does a refactoring induce bugs? An empirical study. In: 12th IEEE Conference on Source Code Analysis and Manipulation, SCAM 2012, Italy, 2012, pp. 104–113 (2012)
3. Briand, L., Devanbu, P., Melo, W.: An investigation into coupling measures for C++. In: International Conference on Software Engineering, vol. 12 (1999)
4. Field, A.: Discovering Statistics Using IBM SPSS Statistics, 4th edn. Sage Publications Ltd., Thousand Oaks (2013)
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co. Inc., Boston (1999)
6. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-finder: a refactoring reconstruction tool based on logic query templates, pp. 371–372 (January 2010)
7. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 552–576. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_23
8. Opdyke, W.: Refactoring: a program restructuring aid in designing object-oriented application frameworks. Ph.D. thesis, Univ. of Illinois (1992)
9. van Deursen, A., Moonen, L.: The video store revisited - thoughts on refactoring and testing. In: Proceedings - XP 2002, Sardinia, Italy (2002)