Empirical Support for Two Refactoring Studies Using Commercial C# Software

M. Gatrell, S. Counsell and T. Hall
Dept of Information Systems and Computing,
Brunel University,
Uxbridge, Middlesex, UK.
mattgatrell@googlemail.com, steve.counsell@brunel.ac.uk, tracy.hall@brunel.ac.uk

Objective. **This paper documents a study of the application of refactorings in commercial C# software comprising 270 versions over a two-month period. The software was developed in a continuous integration environment in a large, multi-national company where each software change committed to the source control was regarded as a new version. The aim of the research was to compare and contrast the results from two previous refactoring studies with those of the C# software.**
Method. **A tool was developed to compare each version with the previous and detect occurrences of fifteen types of refactorings in both production and test classes. In total, over one thousand separate refactorings were identified. We then extended the profile of refactorings to compare (separately) the types of refactorings identified in test code and those in production code. Finally, we explored the inter-relationships between a subset of the fifteen refactorings as a part explanation for the results.**
Conclusions. **While 'simpler' refactorings were common, the more 'complex' structural refactorings were relatively rare. This supported the key result of an earlier empirical study by Advani et al. where Java open-source was used. Distinct differences were found in the types of refactoring applied to each code type. Support was found for a recent observation of Zaidman et al. in terms of parallel effort devoted to production and test classes. The study thus illustrates a strong commonality between refactoring trends found in *both* previous studies. Analysis of, and insight into all of our results were informed by follow-up discussions and consultation with the company's Architect.**

1. INTRODUCTION

Refactoring is the process of improving the internal characteristics of existing software while preserving its external behaviour [10]. Opdyke first documented the process and potential benefits of refactoring in [17], followed later by Fowler's text documenting seventy-two common refactorings [10]. Refactoring has been suggested as a positive influence on the long term quality and maintainability of software and is also said to help prevent (and even reverse) code decay. A typical example of a refactoring is 'Rename Method' where a method is renamed to make its purpose more intuitive (and the code more understandable). While there have been a range of recent empirical studies using refactoring as a basis [1, 5, 9, 19, 20, 22], a broad set of research questions remain unanswered. In particular, limited availability of commercial, proprietary software has restricted our understanding of refactoring trends in this type of system and the comparison that that would allow with open-source software (OSS). Understanding what types of refactorings are undertaken in each and contrasting production code with test code can provide the basis for planning future refactoring effort and the effectiveness of an adopted test strategy.

In this paper, we describe the results from applying a bespoke tool to 270 versions of a large, commercial software product. We examined each version with its predecessor to detect application of 15 types of refactorings in both production and test code classes. The extracted data was analysed to identify characteristics in 1038 separate refactorings extracted by the tool. Results showed a strong parallel between the proprietary code and the results from OSS previously reported by Advani et al. [1]. In this sense, the study supports that earlier work. More specifically, simple refactorings were common; complex 'structural-based' refactorings were less common. A distinct difference between test and production code (in terms of applied refactorings) was found, despite the fact that they shared a strong inter-dependence, namely that changes to production classes usually required corresponding changes to test classes; this result was also recently found by Zaidman et al. [20]. The goals of the study presented are thus first, to determine the extent of similarity with the previous study of refactoring by Advani et al, and, second, those with the study of test versus production classes by Zaidman et al. Our analysis of the data is supported, wherever possible, with input from the Architect of the system studied (with whom follow-up discussions and analysis were conducted).

The remainder of the paper is organised as follows. In the next section, we describe the motivation for the study and related work. In Section 3, we describe preliminaries such as the system studied and the refactorings extracted by the tool. We then analyse the data (Section 4) exploring three facets of the data, before discussing the threats to the validity of the study (Section 5). Finally, we conclude and point to further work in Section 6.

2. MOTIVATION AND RELATED WORK

The research presented is motivated by a number of factors. First, until now, there has been only limited research to determine how refactorings are applied in commercial software (and much of this uses OSS [4, 8]). The authors know of no previous study that has compared OSS with that of proprietary, in-house software. This has been largely due to the lack of access to data from commercial systems. We see an understanding of the differences between the two types of system as a research challenge with significant relevance to industrial practice; increasingly, organisations are using both types of software as part of their development strategy. Second, test code and its development may have specific characteristics and features that distinguish it from production code [2, 3, 7, 11]. This means it may be less, equally, or more suitable for refactoring. Only by contrasting the two types of code can we begin to understand which of those applies and whether the evolutionary characteristics of each support any distinguishing features. The two studies with which we compare our results are that of Advani et al. [1] and Zaidman et al. [20]. The former showed that 'simpler' easily undertaken refactorings were more common than 'complex' structural (more effort intensive refactorings). Our study also draws parallels with the recent study by Zaidman et al. [20], where production and test classes were found to co-evolve in a synchronous way. A Java system was used as a basis of their study and a number of other research questions were posed related to test strategies and test coverage.

In terms of related work, Opdyke introduced the concept of refactoring [17], followed some years later by Fowler's text [10] defining 72 common refactorings. Zhao and Hayes [22] used metrics to attempt to predict and prioritise the classes that were most in need of refactoring. Their tool applied metrics to locate design problems in classes and presented them to the developer in order of priority. They also observed that manual refactoring selection by Java programmers tended to overlook the difficult, more complex and large refactorings and concentrated instead on the easier, smaller refactorings. This view is supported by Advani et al., [1] where refactoring data across multiple versions of the same software was mined to show which refactorings had been applied and when. Research showed the majority of refactorings were simpler refactorings and not complex structural refactorings. Counsell et al., [4] also investigated the most commonly used and least commonly used refactorings and the inter-dependencies of those refactorings; refactorings that featured rarely were those with many inter-dependencies. Commonly applied refactorings were those with fewer inter-dependencies.

In other work, Tokuda and Batory have shown how significant quantitative benefits can be obtained through automated refactoring [19]. Mens and Tourwe [12] provide a comprehensive overview of research into refactoring including types of software artefacts that are subject to refactoring and consideration of the effect of refactoring on the software process; Mens and van Deursen [13] also present emerging trends and open problems in the field of refactoring. Further, van Deursen et al., [7] recognise that test code has different structures and patterns to production code and published test-specific smells and refactorings to reflect this; van Rompaey et al., [18] extended this study to define metrics to detect test smells [4, 10]. Finally, Zeiss et al., [21] have explored which refactorings were applicable to test structures and those applicable to test behaviour (in this study, we attempt to shed light on this feature as well).

3. PRELIMINARIES

The anonymous system used as a basis of the empirical study is from a large, international software company specialising in transaction content processing software. One of the authors works as a Software Architect in the same company and has access to the version control system and hence the changes made to the system during the 270 versions studied. The system relates to a core technology product written in C# by a team of 8-10 developers and had been running for approximately 12 months from when our analysis started; it includes server side components, a web application and a number of client side components and tools. Henceforward, for confidentiality purposes, we will refer to the system as 'WebCSC', comprising over 3000 classes (at the latest version). During the two-month period over which the 270 versions spanned, WebCSC was subject to modifications due to new enhancements as well as fault-fixing. In this paper, we make no distinction between the two types of maintenance modification (although we plan to explore these two aspects in future work). The WebCSC system was developed in a continuous integration environment where every change committed to source control by a developer was explicitly versioned and built by the continuous integration server; this allowed us to compare code at each version step. On average, a version was submitted to the source control system every hour over the period studied. A significant portion of the source code comprised (unit) test classes and we include these as well as production classes as part of our analysis [14].

3.1 Extracted refactorings

The tool attempted to detect occurrences of fifteen specific refactorings (in keeping with the earlier study by Advani et al., [1]). The choice of the fifteen refactorings in that earlier study was based on the views of two industrial developers of the most likely refactorings that developers would undertake as part of their daily maintenance. For

convenience, the order of the fifteen refactorings in the following list is *as per the study* of Advani et al., and represents the least frequently applied refactoring (Encapsulate Downcast) to the most frequently applied (Rename Field):

1. Encapsulate Downcast (ED). According to Fowler [10], 'a method returns an object that needs to be 'downcasted' by its callers'. In this case, the downcast is moved to within the method and the methods return type is changed.
2. Push Down Method (PDM). 'Behaviour on a superclass is relevant only for some of its subclasses' [10]. The method is moved to those subclasses.
3. Extract Subclass (ESub). 'A class has features that are used only in some instances' [10]. In this case, a subclass is created for that subset of features.
4. Encapsulate Field (EF). The declaration of a field is changed from public to private.
5. Hide Method (HM). 'A method is not used by any other class' [10] (the method should thus be made private).
6. Pull Up Field (PUF). 'Two subclasses have the same field'. In this case, the field in question should be moved to the superclass.
7. Extract Superclass (ESup). Two classes have similar features. In this case, a superclass is created and the common features moved to the superclass.
8. Remove Parameter (RP) (from the signature of a method).
9. Push Down Field (PDF). 'A field is used only by some subclasses' [10]. The field is moved to those subclasses.
10. Pull Up Method (PUM). Methods have identical results on subclasses. In this case, the methods should be moved to the superclass.
11. Move Method (MM). 'A method is, or will be, using or used by more features of another class than the class on which it is defined' [10].
12. Add Parameter (AP) (to the signature of a method).
13. Move Field (MF). 'A field is, or will be, used by another class more than the class on which it is defined' [10].
14. Rename Method (RM). A method is renamed to make its purpose more obvious.
15. Rename Field (RF). A field is renamed to make its purpose more obvious.

3.2 Tool mechanics and validation

To extract each of the fifteen refactoring types, the source data comprising class names, namespaces, method names, method signatures, parameters, fields and return types were stored in a database allowing a) multiple source versions to be loaded and compared at the same time and, b) queries to be run looking for the symptoms of a refactoring. For example, one heuristic was that if a field existed in class A in version x, and was removed from class A in version $x + 1$ (and subsequently added to class B in version $x + 1$), an occurrence of a Move Field refactoring was noted. The tool was validated against a set of test classes written specifically to test the output of the tool. Across the test classes, multiple instances of all 15 refactorings were introduced alongside other non-refactoring activity. The test passed if all refactorings were identified by the tool. A sample of the WebCSC classes was then randomly selected to use as input to the tool; this sample was manually inspected to verify that the refactorings identified were correct.

4. DATA ANALYSIS

4.1 Refactoring frequency

During the two-month period incorporating the 270 code versions, a total of 1038 refactorings drawn from the set of 15 refactoring types were detected by the tool from 196 classes. Of the 270 versions, 94 incorporated at least one refactoring; no refactorings were therefore applied in 176 versions. The highest ranked 30 classes from the 196, in *descending* number of applied refactorings, accounted for 51.15% (531) of the total number of refactorings. A high proportion of classes had had only a single or two refactorings applied to them over the course of the 370 versions. Figure 1 illustrates the distribution of this data. The class with the most number of applied refactorings (52) was a test class; in fact only 13 of the highest ranked 30 classes were production classes - test classes underwent relatively larger amounts of refactoring than the corresponding production classes. The explanation for such a high level of refactoring in these classes, supported by the WebCSC Architect was that for every production class refactoring, up to five changes have to be made to the test class responsible for testing that production class.

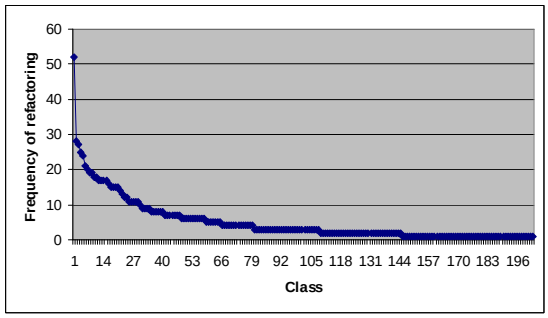


FIGURE 1. FREQUENCY OF REFACTORINGS IN CLASSES

Table 1 shows the frequency breakdown of refactorings. Six classes had had between 21 and 30 refactorings applied and 23 classes had had between 10 and 19 applied refactorings. The high bias towards classes having 1 or 2 refactorings can be seen from the table where 87 classes fell into this category.

>= 30	>= 20	>=10	> 2 < 10	= 1 <= 2
1	6	23	79	87

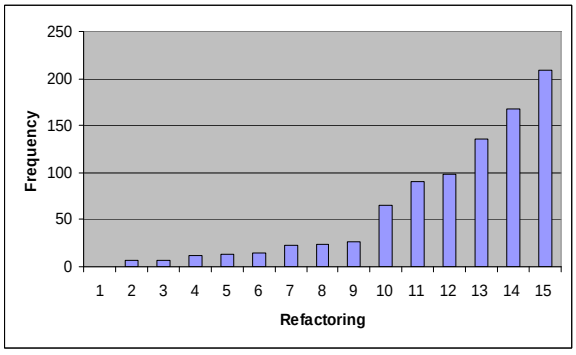
TABLE 1. FREQUENCY OF APPLIED REFACTORINGS

4.2 Analysis themes

The data analysis in this paper is explored through three themes. First, we explore any potential similarities between the number and type of refactorings found in our study and that found in the previous study by Advani et al., [1] where OSS was used as a basis. Second, we explore the emphasis of, and comparison between, refactorings in test code and those in production code to establish whether any significant differences or patterns could be found. Finally, since the tool identified both the class and method where each of the 1038 refactorings were undertaken, we investigate whether specific 'related' refactorings were applied in unison to the same class or method. An open research question at present is whether application of one refactoring implies that at least one other related refactoring also has to be applied at the same time [16].

4.2.1. Study comparison.

Figure 2 shows the data for the fifteen refactorings extracted by Advani et al., [1] from multiple versions of seven OSS in ascending (and same) order on the *x*-axis as per the list given in Section 3.1. For example, the least frequently applied was refactoring 1 (Encapsulate Downcast) and the most frequently applied was refactoring 15 (Rename Field). Figure 3 shows the values extracted for WebCSC, in the same order as in Figure 2.



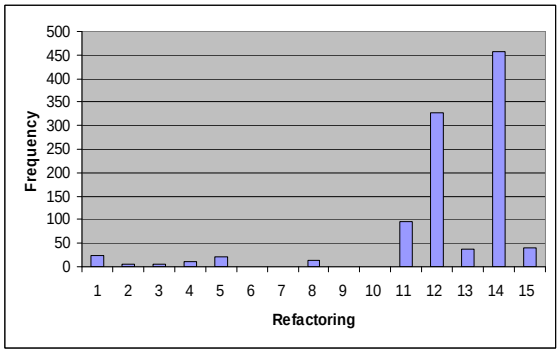


FIGURE 2. REFACTORINGS TAKEN FROM [1]

FIGURE 3. REFACTORINGS (WebCSC SYSTEM)

For the WebCSC system, we see that over the 270 versions studied there was a relatively high occurrence of five individual refactorings (i.e., refactorings 11-15 from Figure 3). When compared with the study by Advani et al., [1], we observe a strong correspondence. In that earlier study, a 'Gang of Six' refactorings were identified (i.e., refactorings 10-15 from Figure 2). Concurring with that earlier study, five of those refactorings in the WebCSC system *also* seem to have been applied relatively frequently. The more 'complex' structural refactorings, on the other hand, such as Extract Subclass (refactoring 3) and Extract Superclass (refactoring 7) both of which require structural changes to the inheritance hierarchy were rarely applied to either the earlier study by Advani et al., *or* to the WebCSC system. In fact, no occurrences of the Extract Superclass refactoring were found in the latter system

and only 4 Extract Subclass refactorings were found in total across all 270 versions. For the WebCSC system, 25 Encapsulate Downcast refactorings were identified by the tool, but in the earlier study by Advani et al., not a single one of these refactorings was identified in any version of the seven systems. Inspection of the raw data revealed the majority of the ED refactorings to occur exclusively in production classes (rather than test classes).

The most common refactoring from Figure 3 is refactoring 14, Rename Method (RM). Surprisingly, the whole set of RM refactorings occurred in only 50 of the 270 versions (i.e., a large number of renaming of methods occurred in a comparatively small number of classes). In version 154, 201 of the total of 458 RM refactorings were found to have been applied. Figure 4 shows the profile of the RM refactoring across the versions of the WebCSC system; the flattening trend at version 154 is clearly visible and striking. Consultation with the WebCSC Architect revealed why version 154 comprised so many RM refactorings. The work being undertaken on the set of classes in which those RM were applied had been 'checked out' and had been worked on for a relatively longer period of time than the set of classes changed in other versions. In other words, the high number of RM refactorings reflected the amount of time that the relevant classes had been 'out of the system'. It is also interesting that nine of the 23 Encapsulate Downcast refactorings for the WebCSC system shown in Figure 3 were applied in version 154. This was due to the large number of renaming methods in the same version, the signatures of which were affected when an ED refactoring was undertaken.

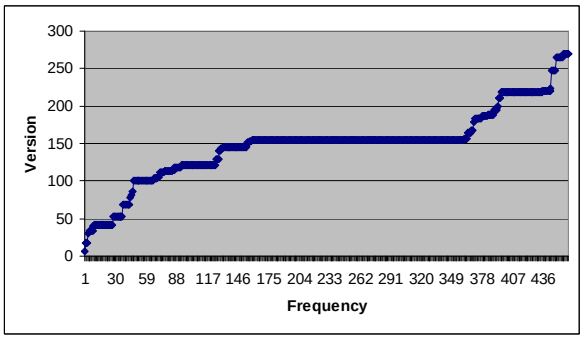


FIGURE 4. PROFILE FOR RENAME METHOD

Figure 3 also shows that 328 Add Parameter (AP) refactorings were undertaken over the course of the 270 versions. Figure 5 shows the version profile for this refactoring and indicates a relatively consistent application of AP over the course of the 270 versions (in contrast to the graph in Figure 4 where version 154 experienced a relatively large number of RM refactorings). Addition of parameters is a standard change made frequently to test classes in the face of regular changes and addition of functionality to the set of production classes. Consequently, the AP refactoring was frequently and regularly applied to test classes. Typically, a WebCSC test class includes test methods for testing a) every parameter in the production class b) parameter exception conditions (for example, null checks) c) outcomes of the method d) every exception condition and e) every execution path that has not already been tested. Multiple changes to a production class usually induce multiple changes to the corresponding test class.

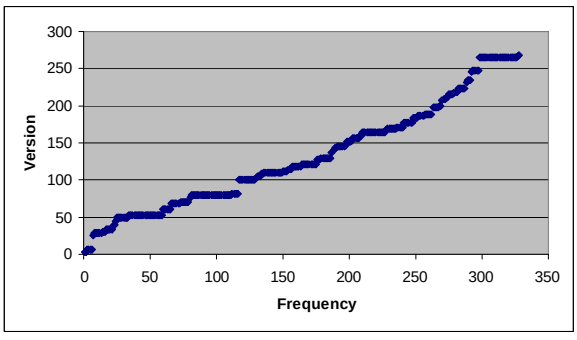


FIGURE 5. PROFILE FOR ADD PARAMETER

4.2.2 Trend analysis.

A feature of the earlier analysis by Advani et al., [1] was the tendency for significant refactoring activity to occur, followed by relatively quiet periods of inactivity. Figure 6 shows the analysis for WebCSC for the first fifty versions and Figure 7 the number of refactorings for versions 51-200. There is a clear trend for refactorings to increase steadily between versions 1 and 40, and moreover, in an erratic fashion. The peak of 16 refactorings at version 42 comprised entirely of Move Method (MM) and RM refactorings, unique to a set of test classes. In the case of the same 16 refactorings, visual inspection of the data revealed that many methods *moved* were those that were then

subsequently *renamed*. In the WebCSC system, if a production class method is moved and renamed, then this requires the corresponding test class method also be moved and renamed to retain the 'semantic' mapping.

Figure 7 shows the refactoring data for versions 51 to 200 (sample size 1-150) and shows a similar effect to that in Figure 6. Remarkably, from Figure 6, the average number of refactorings *per* version (sample 1-50) was 1.92, compared with 5.34 *per* version from Figure 7. The average then fell to 2.1 between versions 201 and 269 (Figure 8). In other words, the earlier versions of the system saw relatively small, yet fluctuating amounts of refactoring and so too did the later versions; the interval in between however saw a large rise in refactorings. In terms of numbers, between versions 1-50 there were only 92 refactorings; that rose to 801 in versions 51-200 and 145 in versions 201 to 270. (Note: we have chosen these version ranges for clarity of presentation purposes only.)

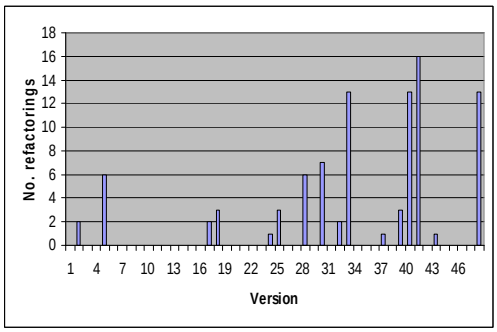
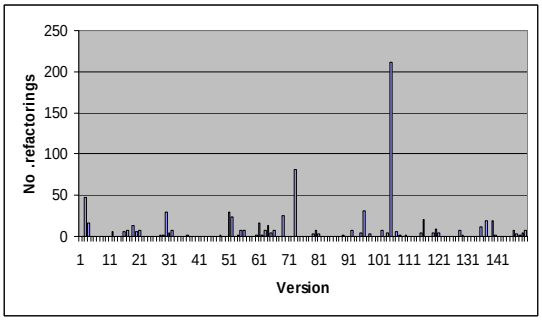


FIGURE 6. REFACTORING ACTIVITY FOR WebCSC (FIRST 50 VERSIONS)



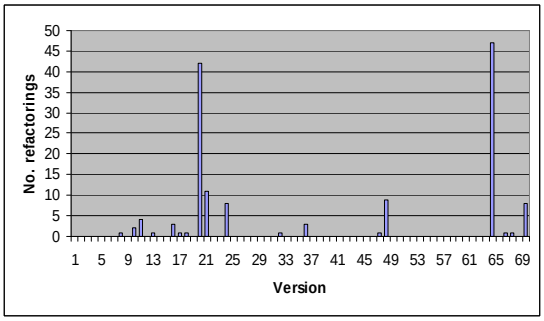


FIGURE 7. REFACTORING (VERSIONS 51-200) FIGURE 8. REFACTORING (VERSIONS 201-270)

One question that arises from the preceding analysis is whether, during the three stated periods (and particularly the middle), the system saw a rapid growth in either number of classes or methods. We would usually expect a relatively high rise in number of refactorings to accompany a corresponding addition of functionality (classes and methods) to a system. Between versions 1 and 50, only 21 classes were added to the system; between versions 51 and 200, 167 classes were added and between versions 201 and 270, only 35 classes were added. This suggests that addition of classes may have been the impetus for extra required refactoring effort noted in the middle interval in this case. We also note that the corresponding three periods saw a rise of 215, 1177 and 225 methods, respectively - further supporting the view that a relatively large increase in classes and associated methods induced correspondingly large numbers of refactorings. In terms of the first goal of the study, we therefore found strong parallels between the study of Advani et al. [1] and that of the study presented.

4.2.3. Test versus production code.

The tool identified refactorings in both test code *and* production code. One question that we explored was the extent to which refactorings (and composition of refactorings) in each of these two types of code differed (or concurred) in keeping with the study of Zaidman et al. [20]? Of the 3174 classes in the system at version 270, 1735 (54.6% of the total) were production classes and 1441 (45.4%) test classes. Based on discussions with the WebCSC Architect, corresponding and concurrent fluctuations in refactorings could be expected to have been applied to both test classes and production classes, since changes in one are usually reflected in required changes to the former. We would also expect the *type* of refactoring in each case to be similar. In the case of both production and test code, we would expect a high proportion of movement and renaming (MM and Rename Field (RF) and RM), as well as many AP refactorings to reflect the need to test features of modified production classes.

4.2.4. Numbers of refactorings.

Figure 9 shows the number of refactorings of the fifteen types in the test classes compared with the same fifteen in the production classes. (The order of the refactorings is the same as that in Figures 2 and 3.) Table 2 shows the actual values taken from this figure.

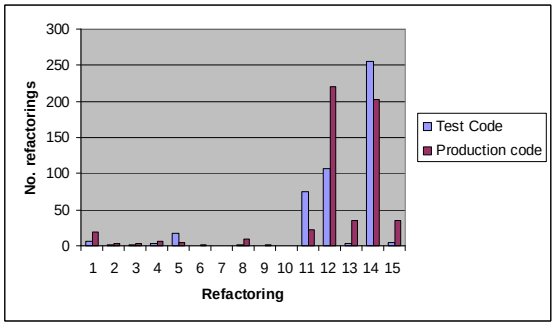


FIGURE 9. REFACTORINGS (TEST AND PRODUCTION CODE)

The notable feature of Figure 9 (and Table 2) is the similarity of the RM refactorings (refactoring 14) in the test code (255) compared with those in the production code. The explanation for this feature is that a single renaming in a production class often requires at least one renaming in the class that actually tests the production class. If a production method changes significantly, then the names of the methods that test the production method have to be modified accordingly to reflect the modified purpose of the method. A notable feature of Figure 9 is the unexpectedly high value of the AP refactoring in the production classes (refactoring 12). A single change made to the signature of a production class means that a corresponding test class may have to be changed to test new behaviour in the production class. It appears that this was not necessarily the case. This feature of the data may highlight the difference between refactorings necessary as a result of fault fixes and those as part of code enhancements. Some occurrences of production refactorings were in response to faults uncovered by the test classes which only required changes to production code and subsequent re-test. Faults in test classes posed a particularly difficult problem, since the integrity of the production code is severely compromised as a result.

TABLE 2. REFACTORINGS (TEST AND PRODUCTION CLASSES)

	Refactoring	**Test**	**Prod.**
1.	Encapsulate Downcast	6	19
2.	*Push Down Method*	1	3
3.	*Extract Subclass*	1	3
4.	Encapsulate Field	3	7
5.	Hide Method	17	4
6.	*Pull Up Field*	0	1
7.	*Extract Superclass*	0	0
8.	Remove Parameter	2	10
9	*Push Down Field*	0	1
10.	*Pull Up Method*	0	0
11.	Move Method	75	22
12.	Add Parameter	107	221
13.	Move Field	3	35
14	Rename Method	255	203
15.	Rename Field	4	35

The general lack of inheritance-based refactorings (all of which are italicised in Table 2) is a feature of both test and production classes. The absence of inheritance-based refactorings in test classes is because test classes are usually independent, self-contained units and have a one-one mapping between with the production classes they test. As such and unlike production classes, there is no requirement or motivation to either inherit behaviour or take advantage of inheritance features from other test classes.

The relatively large number of Hide Method refactorings in the test code is an exceptional situation and is the result of a single developer using temporary methods in a test class which were then changed from public to private once those methods had been used (the system's Architect confirmed this to be the case). One final feature of the set

of test class refactorings is that there are relatively few RF refactorings (refactoring 15), in complete contrast to production classes. A simple explanation accounts for this feature supported by the follow up discussions with the Architect; in test classes, very few fields (i.e. attributes) are generally declared, since the class has no obvious need to store or manipulate data. The importance of refactoring test classes is signified by the fact that overall, 474 of the 1038 refactorings were found in the test code (approx. 46% of the total).

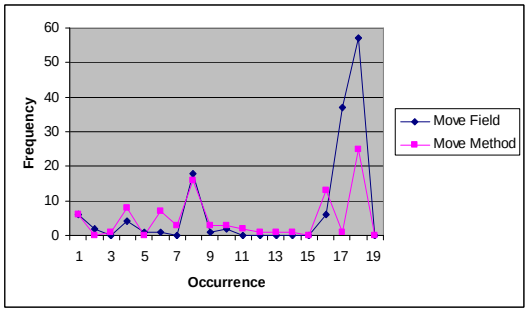
4.2.5. Version analysis.

A further aspect of any difference between test code and production code is to what extent the application of refactorings were applied in the test code and production code *in the same versions*. In other words, when a refactoring was applied to production code, was at least one refactoring applied to test code? The preceding discussion implies that the two may go hand-in-hand. To investigate, we computed the number of versions where a) there was at least one test code refactoring *and* at least one production code refactoring, b) there were just production code refactorings and finally, c) where there just test code refactorings.

The data revealed that in 34 versions there was: 'at least one test refactoring *and* one production class refactoring'; in 46 versions only production code refactorings were undertaken and finally, in 14 versions, only test class refactorings were undertaken. The majority of the versions where there were only production class refactorings tended to be single, isolated refactorings. This contrasts with occasions when only test class refactorings were applied, when there was usually a cluster of the same type of refactoring. While there is some evidence to suggest that test and production refactorings are combined in certain versions, there is also evidence of an independence of each type of refactoring (given by the fact that they were applied unilaterally in different versions). One interesting observation from the data that might explain this feature was that a large number of test classes were refactored in the version *immediately following* a large group of production class refactorings. In other words, developers may, in some cases, apply refactorings to production classes and then, in the next version, make the necessary test class refactorings. This therefore supports the result of Zaidman et al. [20] and this tendency of developers in the WebCSC system was supported by the Architect in the follow-up discussions.

4.2.6. Related refactorings.

One aspect of refactoring that we also wanted to explore was the tendency for a combination of different types of refactorings to be applied at each version. For example, if we move a method, then we might reasonably expect to move a field at the same time. Equally, we might also expect a method to be renamed after moving it. The mechanics specified by Fowler identify the relationships between refactorings and these relationships have been previously explored in [6]. Figure 10 shows the graph of number of Move Method refactorings on a version-by-version basis and the number of Move Fields when at least one Move Method was identified from the earlier study of Advani et al., [1]. Figure 11 shows the corresponding data for refactorings from the WebCSC system where a strong correspondence between the two graphs can be seen; the relationship is thus preserved in the WebCSC system.



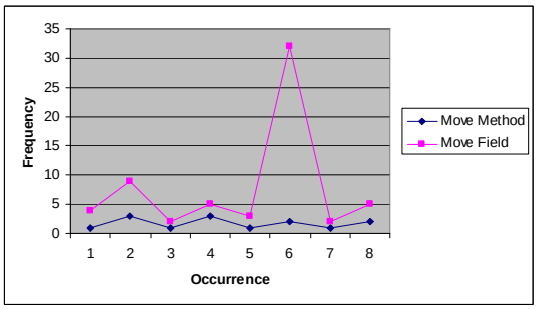


FIGURE 10. RELATIONSHIP - MF and MM [1] **FIGURE 11. RELATIONSHIP - MF and MM (WebCSC)**

A further relationship we might expect is that between RF and RM. Figure 12 shows the relationship extracted from the data for every occasion where 'at least one RF and at least one RM were identified' in the data extracted by the tool. Only on 12 occasions were both RM and RF applied in the same version. Although a strong parallel between the two refactorings exists, the sum of RM refactorings from Figure 12 is 78 from a total of 458; the sum of RF on the other hand was 24 from a total of 39. In other words, RF is more closely tied to RM than *vice versa*. One reason for this may be the existence of *get* and *set* methods in the respective classes and where we might expect the relationship to be unidirectional only.

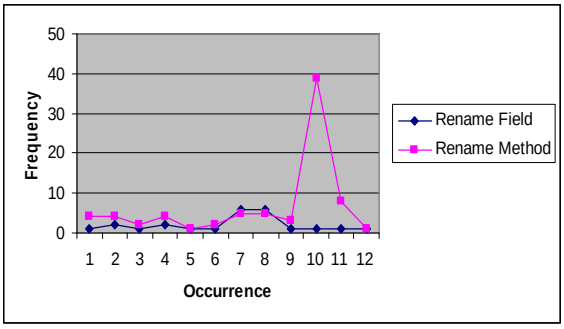


FIGURE 12. RENAME FIELD AND RENAME METHOD

5. THREATS TO STUDY VALIDITY

A number of threats to the validity of the study need to be considered. First, we have only considered fifteen of the seventy-two refactorings listed by Fowler in his seminal text [10]; it is possible that many more refactorings could have been applied over the versions of WebCSC studied. However, the fifteen refactorings chosen represent a broad cross-section of those seventy-two covering OO themes such as inheritance, encapsulation as well as standard tasks that we would expect a developer to undertake as part of their programming (i.e., moving and renaming class features). To implement all seventy-two refactorings is a task that would require development by a competent programmer over many years. We see our study as the 'first-step' towards understanding the intricacies of the refactoring process across environments.

Second, the WebCSC language was written in C# and the study upon which we based our work was written in Java, comprising multiple systems. Further, the environment of WebCSC was proprietary and not OSS, the development process was continuous integration and the time frame for C# system was only one year. In defence of these potential criticisms however, only minor differences exist between Java and C#. A major motivation for the work was to determine if trends in OSS refactoring occurred in proprietary software *and* OSS. It is exceptionally difficult to obtain free-to-use, evolution-based, commercial software and while we do not claim the work to be a replication in the truest sense of the word, we feel that it is important that results from previous studies are tested, especially for an emerging discipline like refactoring. We also feel that it is important to evaluate different environments (proprietary versus OSS) even if the development processes are different (i.e., these comparison problems will *always* exist). Moreover, we could claim that while OSS development at a generic level underpins an ethos of shared and freely-available software, the processes and dynamics of development and people within each OSS system are likely to be very different (itself a threat).

Third, we have reported a similar result to that of Advani et al [1] and to a lesser extent that of Zaidman et al [20], and we have provided an explanation of why that phenomenon might occur due to the relationships between refactorings (Section 4.2.6). A wider issue is whether developers 'consciously' undertake refactorings as part of a concerted refactoring effort or they undertake those refactorings as part of more regular maintenance activity (i.e., it happens automatically when fault-fixing). We see this as a topic of future research. Fourth, we could be criticised for writing a bespoke tool instead of reusing the original one of Advani et al. However, we *did* reuse the algorithms for detecting each of the fifteen refactoring. The original tool mined OSS resources and was geared towards extraction of code and subsequent XML development from sourceforge.net. On the other hand, the data used by WebCSC was already available in an easily converted format. On balance, it would have taken just as long to adapt the original tool than to have re-written it. Fifth, one plausible reason why so few 'complex' refactorings were unused is simply that the majority of time in any system is spent fixing faults or adding new functionality and not on refactoring; equally, that the developers were ignoring OO sound practice. Finally, we accept that *regular* changes to the code in WebCSC (aside from the identified refactorings) are just as interesting and worthy of study and comparison with the refactorings identified. However, we leave that as a significant topic for future work.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have described a study of 1038 refactorings extracted from versions of a large commercial C# system. Fifteen specific types of refactoring were extracted using a bespoke tool. A number of results emerged from the analysis. First, while simpler refactorings were common, more complex structural refactorings were rare in common with a previous study by Advani et al. [1]. Second, refactorings were applied sporadically, rather than consistently over the period of time studied again supporting the earlier study. We also found some interesting similarities between test classes and production classes in support of recent work by Zaidman et al. [20]. Finally, we suggested a reason why certain refactorings might be applied in unison.

The question that is often asked is whether developers *do* refactoring and if so what are the common types of refactoring? The study presented shows that significant amounts of refactorings had been undertaken. Second, an emerging research area is the refactoring requirements for test suites and the set of refactorings that lend

themselves to the way test classes are constructed; we feel we have provided some insights into that area, although we accept there are many more outstanding challenges.

As well as providing support for some research questions, the study also raises a number of other issues. In terms of future work, the work could be extended by increasing the number or refactoring types detected by the tool as well as increasing the number of versions studied; the WebCSC system is an ongoing and ever-developing artefact and from an evolutionary perspective it would be interesting to observe whether the same trends recur as the system ages further. We could also introduce the detection of 'code smells' [10] and their relationship to subsequent refactorings to determine whether there is a link between classes that were heavily refactored with either the fault or change propensity of a class. One other avenue for future work would be to compare different states of the WebCSC application (e.g., after four months with that after eight months, etc for any commonly occurring trends and links with refactorings). Finally, the use of patterns in both test and production code and exploring the change-proneness of classes incorporated into those patterns are also planned for future research. To support further support or refutation of the work presented in this paper, the tool used to extract the data for this study can be made available upon request of the authors.

REFERENCES

[1] D. Advani, Y. Hassoun and S. Counsell. Extracting Refactoring Trends from Open-source Software and a Possible Solution to the 'Related Refactoring' Conundrum. Proceedings of ACM Symposium on Applied Computing, Dijon, France, April 2006.
[2] K. Beck, E. Gamma, Test infected: Programmers love writing tests, Java Report 3(7) (1998) 51–56.
[3] K. Beck, Test Driven Development: By Example, Addison-Wesley, 2003.
[4] S. Counsell, Y. Hassoun, G.Loizou, R.Najjar: Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.
[5] S. Counsell, Y. Hassoun, R. Johnson, K. Mannock and E. Mendes. Trends in Java code changes: the key identification of refactorings, ACM 2nd International Conference on the Principles and Practice of Programming in Java, Kilkenny, Ireland, June 2003.
[6] A. van Deursen and L. Moonen. The Video Store Revisited - Thoughts on Refactoring and Testing. Proceedings of the third International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP 2002, Sardinia, Italy.
[7] A. van Deursen, L. Moonen, A. Bergh, G. Kok, Refactoring test code, in: M. Marchesi (Ed.), Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP 2001), University of Cagliari, 2001, pp. 92–95.
[8] T. Dinh-Trong and J. Bieman. Open Source Software Development: A Case Study of FreeBSD. Proc. 10th IEEE Int. Symp on Software Metrics, Chicago, USA, 2004, pages 96-105.
[9] 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.
[10] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
[11] M. Galli, M. Lanza, O. Nierstrasz: Towards a Taxonomy of SUnit Tests. Proceedings of ESUG Research Track, 2005.
[12] T. Mens and T. Tourwe, A Survey of Software Refactoring, IEEE Trans. on Software Eng. 30(2):126-139 (2004).
[13] T. Mens and A. van Deursen. Refactoring: Emerging Trends and Open Problems. Available online: http://www.swen.uwaterloo.ca/~reface03/Papers/TomMens.pdf
[14] G. Meszaros, xUnit Test Patterns: Refactoring Test Code, Addison-Wesley, 2008.
[15] R. Najjar, S. Counsell and G. Loizou. Encapsulation and the vagaries of a simple refactoring: an empirical study. Proc. Intl. Conference on Soft. Systems Eng. and its Applications, Paris, France, 2005.
[16] M. O'Cinneide and P. Nixon. Composite Refactorings for Java Programs. Proceedings of the Workshop on Formal Techniques for Java Programs. ECOOP Workshops 1998.
[17] W. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, Univ. Illinois at Urbana-Champaign, 1992.
[18] B. van Rompaey, B. Du Bois, S. Demeyer and M. Rieger. On the detection of test smells: A Metrics-based Approach for General Fixture and Eager Test. IEEE Transactions on Software Engineering, 33(12): pp 800-817, December 2007.
[19] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. Automated Software Engineering, 8:89-120, 2001.
[20] A. Zaidman, B. Van Rompaey, S. Demeyer and A. van Deursen: Mining Software Repositories to Study Co-Evolution of Production & Test Code. International Conference on Software Testing (ICST): pp 220-229, 2008.
[21] B. Zeiss, H. Neukirchen, J. Grabowski, D. Evans and P. Baker: Refactoring and Metrics for TTCN-3 Test Suites. 5th Workshop on System Analysis and Modelling (SAM), Kaiserslautern, Germany, 2006, pages 148-165.
[22] L. Zhao and J. Hayes. Predicting Classes in Need of Refactoring: An Application of Static Metrics. 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA (co-located with the IEEE Conference on Software Maintenance), 2006.