Refactoring trends across *N* versions of *N* Java open source systems: an empirical study

Deepak Advani and Youssef Hassoun,
School of Computer Science, Birkbeck, University of London,
Malet Street, London, WC1E 7HX
Email: yhassoun@dcs.bbk.ac.uk
Steve Counsell
Department of Information Systems and Computing, Brunel University,
Uxbridge, Middlesex. UB8 3PH.
Email: steve.counsell@brunel.ac.uk

Abstract

In the past few years, refactoring has emerged as an important consideration in the maintenance and evolution of software. Yet very little empirical evidence exists to support the claim about whether developers actively undertake refactoring, or whether as Fowler suggests that the benefits of doing refactoring are not short-term but too 'long-term' [8]. In this paper, we describe an empirical study of multiple versions of a range of open source Java systems in an attempt to understand whether refactoring does occur and, if so, which types of refactoring were most (and least) common. Fifteen refactorings were chosen as a basis (on seven Java systems) and the code analysed using an automated tool. Results confirmed that refactoring did take place, but the majority were of the simpler, less complex type. Interestingly, the most common refactorings empirically identified were those which, according to Fowler (and from a dependency graph of the 'seventy two' original refactorings), were central to larger more involved refactorings. One conclusion from the study is thus that developer time and effort for relatively large restructuring and testing of refactored code is prohibitive; making small, simple changes is preferred. A further conclusion from our study is that refactoring didn't occur in the earliest or latest versions of the systems we investigated.

1. Introduction

One of the key software engineering disciplines to emerge over recent years is that of refactoring [8, 15, 19]. Refactoring can be defined as a restructuring of the internal structure of a software artefact without changing its external behaviour; Fowler has likened refactoring to the reversal of software decay, in the sense that it repairs badly damaged software. While the outcome of refactoring effort is desirable, there is very little empirical basis to answer the simple question: do developers generally refactor? Since a large proportion of development time is devoted to maintenance, understanding how software is 'changed' over time is of enormous value. Moreover, if the answer to this question is 'yes', then it would be useful to know which types of refactoring are the most common and which the least common. An impression of likely future demands and refactoring trends may then be possible. Anecdotal evidence suggests that developers have very little time to devote to larger code restructurings often involving an inheritance hierarchy.

In this paper, we describe the results of an empirical study of the trends across multiple versions of open source Java software. A specially developed software tool extracted data related to each of fifteen refactorings from multiple versions of seven Java systems according to specific criteria. Results showed that, firstly, the large majority of refactorings identified in each system were the simpler, less involved refactorings. Very few refactorings related to structural change involving an inheritance relationship were found. Secondly, and surprisingly, no pattern in terms of refactorings across different versions of the software was found. Results thus suggest that developers do simple 'core' refactorings at the method and field level, but not as part of larger structural changes to the code (i.e., at the class level). It is unlikely that we will be able to identify whether those 'core' refactorings were done in a conscious effort by the developer to refactor, or as simply run-of-the-mill changes as part

of the usual maintenance process. However, we feel that identification of the major refactoring categories is a starting point for understanding the types of change and the inter-relationships between changes typically made by developers.

In the next section, we describe related work. In Section 3 we describe the refactorings extracted, the systems used and the criteria adopted by the tool to extract the refactoring data. In Section 4 we discuss the data extracted using the tool and use the data to examine three suppositions. We then discuss some of the issues raised from this study (Section 5) and conclude, pointing to future work, in Section 6.

2. Motivation and Related Work

The motivation for the study in this paper stems from a number of issues. Firstly, there has been a large amount of interest in the criteria for carrying out refactoring [6]. In other words, the decision as to when certain types of refactoring should be undertaken. Yet very little empirical data addresses the question of how widespread refactoring is in practice. The results in this study support earlier findings from an empirical study of a set of library classes [5]. In that paper, the 'substitute algorithm' refactoring [8] (i.e., modification of the body of a method to improve the way it functions) together with the core refactorings investigated herein were found to be the most popular type of change identified.

Secondly, an open research issue is whether refactorings are compound in nature. Does one refactoring always require specific types of other refactoring (empirically speaking)? In this paper, we use a dependency diagram of the seventy-two refactorings to determine whether empirical relationships between refactorings match the theoretical relationships. For example, if refactoring X insists on carrying out refactoring Y first, does the empirical data reflect these dependencies?

Finally, we would expect changes of any type to grow over the lifetime of the system. So we would expect there to be clear (increasing) refactoring trends as a system evolves. Yet, if a system is refactored frequently, then in theory it does not need to have increasing amounts of maintenance applied to it and 'peak' and 'trough' patterns should appear. A key motivation is therefore to see if the trends in refactorings follow any specific patterns as the system evolves. The need for more studies into software evolution issues is highlighted in Perry [18].

In terms of related work, the seminal text and from which our fifteen refactorings were taken is that of Fowler [8]. The work of Opdyke [15], Johnson and Foote [10] and Johnson and Opdyke [16] has also been instrumental in promoting refactoring. Earlier work by Najjar et al. has shown the quantitative and qualitative benefits of refactoring [12]; the refactoring 'replacing constructors with factory methods' of Kerievsky [11] showed quantitative benefits in terms of reduced lines of code and potential qualitative benefits in terms of improved class comprehension. Developing heuristics for undertaking refactorings based on system change data has also been investigated by Demeyer et al. [6].

In terms of automating the search for refactoring trends, research by Tokuda and Batory [19] has shown that three types of design evolution, including that of hot-spot identification, are possible. A key result of their work was the automatic (as opposed to hand-coded) refactoring of fourteen thousand lines of code. Finally, the principles of refactoring are not limited to object-oriented languages; other languages have also been the subject of refactoring effort [2].

The findings in this study suggest that refactorings based on inheritance are infrequently made. It may be that developers avoid any restructuring inheritance hierarchies because of the relatively large number of class dependencies (i.e., coupling) and the subsequent testing effort required. A number of studies have investigated inheritance and cast doubt on the way that inheritance is used in practice [3, 9], thus supporting the view that inheritance-based refactorings are avoided by developers. Finally, very little research has been carried out into composite refactorings [14], where one refactoring is followed by *n* other refactorings.

3. Study Details

3.1 The fifteen refactorings chosen

The choice of which fifteen refactorings to implement in our tool was based on two criteria. Firstly, on the likelihood of finding large numbers of those refactorings over versions of the systems. This led us to implement simple refactorings such as those found to be common in single versions of the library classes of an earlier study [5]. Secondly, we wanted to see if more involved (i.e., complex) refactorings were undertaken and on what scale. We thus implemented the search for a set of refactorings requiring structural changes to the system to be made; for example, those related to an inheritance hierarchy. All the refactorings apart from refactoring number 9 (Rename Field) were taken from Fowler's text. The fifteen refactorings

chosen and the circumstances motivating that refactoring (added where not obvious) were:

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

Fowler divides refactorings into different groups depending on the activity employed in transforming the system. According to Fowler’s classification, our refactorings are chosen from the groups:

1. Making Method Calls Simpler: refactorings 1, 2, 3, 4, 5.
2. Organising Data: refactoring 6.
3. Moving Features Between Objects: refactorings 7, 8.
4. Dealing with Generalisation: refactorings 10, 11, 12, 13, 14, 15.

We note that in the case of certain refactorings, use of our software tool to assist was impossible unless the semantics of the code change were investigated. For example, the ‘substitute algorithm’ refactoring where one or more lines in the body of a method are changed would require the tool to check every line in every method in every class for a single change in the body of that method; even then it would require certain assumptions to be sure of the scope of change. For systems with thousands of classes in each of n versions, the problem this poses becomes clearer. The same problem arises with the extract method refactoring where one method is split into two (to become two methods). The parser, an integral part of our tool [1], would have to check groups of lines of code in any new methods added (to a later version) with all lines of code in methods of the earlier version.

Each refactoring transformation that we implemented was defined through a set of rules or criteria. For example, to detect whether the ‘Move Field’ refactoring had taken place in the transition from one release to the next, the tool checked whether:

1. A field (name, type) that appeared in a class type (belonging to older version) appeared to be missing i.e., has been dropped from the corresponding type of a later version.
2. The field (name, type) did not appear in any superclass or subclass of the original type.
3. A similar field (name, type) appeared to have been added to another type (belonging to later version).

If the criteria (1 to 3) was satisfied for the field under investigation, the tool reported an occurrence of the Move Field refactoring. Similarly, the criteria for ‘Extract Superclass’ was as follows:

1. A class type whose ‘unaccounted’ for fields and/or methods have been pulled up into a newly created superclass (that did not exist in a former release) to become a superclass.
2. The class from which this superclass was extracted has become the base type.

The criteria for the ‘Encapsulate Field’ refactoring is simply that a field in a later version in the same class, and with the same name, has had its declaration changed from public to private. To verify that the data

produced by the software tool was correct, manual checks of the source code (from which the refactoring data was selected) were carried out by the authors.

3.2 Java systems chosen

Seven open source Java systems were analysed as part of our study. We note that we included both classes and interfaces in our analysis.

1. MegaMek. A computer game. The number of classes and interfaces in this system remained static at 190 and 13, respectively.
2. Tyrant. A graphically-based, fantasy adventure game. Incorporates landscapes, dungeons and towns. The system began with 112 classes and 5 interfaces. At the tenth version, it had 138 classes and 6 interfaces.
3. Velocity. A template engine allowing web designers to access methods defined in Java. Velocity began with 224 classes and 44 interfaces. At the tenth version, it had 300 classes and 80 interfaces.
4. Antlr. Provides a framework for constructing compilers and translators using a source input of Java, C++ or C#. Antlr began with 153 classes and 31 interfaces. The latest version had 171 classes and 31 interfaces.
5. HSQLDB. A relational database application supporting SQL. HSQLDB started with 52 classes and 1 interface. The latest version had 254 classes and 17 interfaces.
6. JasperReports. A Java reporting tool to help produce page-oriented documents in a simple and flexible way. JasperReports started with 288 classes and 50 interfaces; the latest version comprised 294 classes and 52 interfaces.
7. PDFBox. A Java PDF library allowing access to components found in a PDF document. The initial system had 135 classes and 10 interfaces; the latest version had 294 classes and 52 interfaces.

3.3 Description of the tool

The set of values for an entire system are represented as an XML tree consisting of sequences of sub-trees representing the individual types. Using this representation, XML data which appears to have been refactored can be identified by applying the criteria described in the previous section. The tool compares consecutive releases of the same industrial software system according to that criteria.

Figure 1 displays the functionality of the tool in sequential order of execution. Phase-1 produces an XML document file in one step for each release of each system. When two consecutive releases of a system are parsed into XML files, phase-2 is initialised. In phase-2, consecutive releases of all systems are compared (one by one). Once all the consecutive releases of each and every API have been compared, phase-3 is initialised. Phase-3 gathers statistics about the refactorings performed. For the interested reader, a detailed description of our tool is provided in [1].

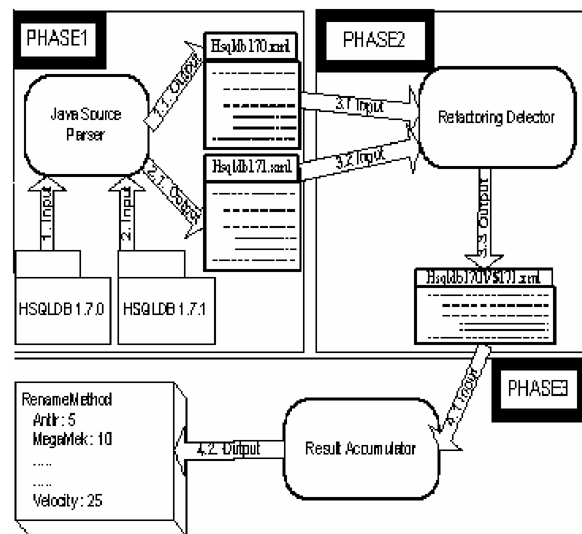

Figure 1: Structure of the refactoring tool

4. Data Analysis

We investigated three suppositions in our analysis of the data. Firstly, we investigated the question of which are the most and least common refactorings across all versions of the systems studied. Secondly, we investigate whether, within each of the seven systems, any refactoring trends are evident. Thirdly and finally, we investigate whether there are any patterns in refactoring across versions of the systems investigated and analyse the possibility that refactorings are connected (in the sense that one refactoring always follows another specific type of refactoring).

4.1 Supposition One

Our first supposition examines which refactorings are the most common from the systems analysed and

equally, which are least common. Prior to the study, we had no pre-conceived ideas about the most likely findings of the tool. A reasonable assumption might have been that the more 'involved' refactorings would be less frequent because of the extra work involved on the part of the developer. Figure 2 shows the frequency of refactorings uncovered by the tool in the form of a bar chart. The order of the bars on Figure 2 follows the order of the legend (i.e., for MegaMek, then JasperReports etc.)

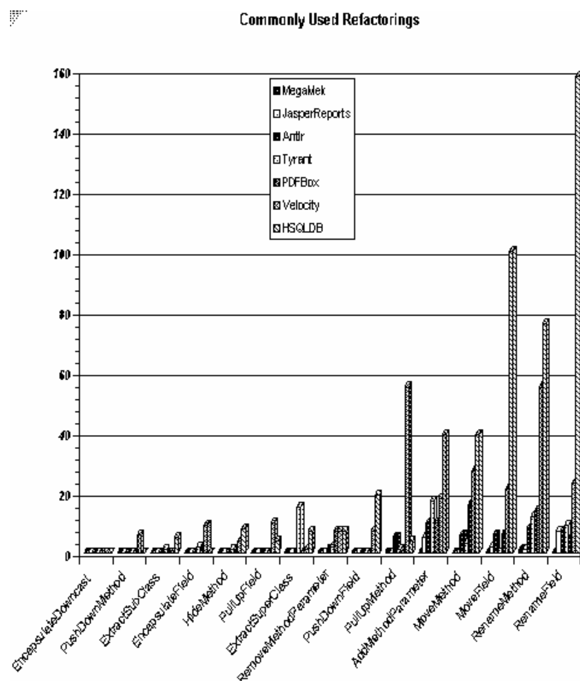

Figure 2: Refactorings extracted from the seven systems

Figure 2 shows the most popular refactoring to be the 'Rename Field' refactoring, followed by 'Rename Method'. In the HSQLDB system, 158 occurrences of this refactoring were found. Interestingly, for the MegaMek system, no occurrences of this refactoring were found.

Least popular was the 'Encapsulate Downcast' refactoring (zero occurrences were found across the seven systems) and the 'Push Down Method' refactoring (only 6 occurrences, all for the Velocity system). The 'Pull Up Method' refactoring value for Velocity is noteworthy since 55 occurrences of this refactoring were found (we investigate this feature of Velocity in Section 4.2).

One surprising result from Figure 2 is the lack of 'Hide Method' and 'Encapsulate Field' refactorings. In theory, these are quite simple refactorings; in each case, the mechanics of doing each of these refactorings are trivial. For hide method, the declaration of the method is changed from public to private. For the encapsulate field, an identical process is carried out. A number of suggestions could be made for the low numbers of these two refactorings. Firstly, we suggest that developers do tend to attach a low priority for visibility issues in terms of declaration of methods and attributes. We support this claim with earlier work on five C++ systems, where trends in encapsulation showed anomalies in four of the five systems. In particular, we found protected attributes in classes without any inheritance coupling [4]. Secondly, recent work by Najjar et al. [13] has found that even for a simple refactoring such as encapsulate field, a number of problems arise. These related to high coupling of the field and problems with the position of the class in the inheritance hierarchy (and dependencies thereof). Practically, simple refactorings are not always that simple.

Although we would not have expected a high number of inheritance-based refactorings, the exceptionally low value found was a surprising result. Perhaps it is the nature of open-source software (where independent developers can make unilateral changes) that explains why changes requiring developers to re-organise the system's structure do not take place. More studies would be needed before any conclusion could be drawn on this issue. Table 1 shows the aggregate refactorings for the seven systems together with the maximum, minimum and mean value for each refactoring over the seven systems.

Table 1: Refactoring summary data

Refactoring Type	Max	Min	Mean	Total
Encap. Downcast	0	0	n/a	0
PushDown Meth.	0	6	0.86	6
Extract Subclass	0	5	0.86	6
Encapsulate Field	0	9	1.71	12
Hide Method	0	8	1.86	13
Pull Up Field	0	10	2.00	14
Extract Superclass	0	15	3.29	23
Remove Paramet.	0	7	3.43	24
Push Down Field	0	19	3.71	26
Pull Up Method	0	55	9.29	65
Add Parameter	0	39	14.14	99
Move Method	0	39	13.00	88
Move Field	0	100	19.29	135
Rename Method	1	76	23.86	167
Rename Field	0	158	29.86	209

The key result from the data shown in Figure 2 (and evident from Table 1) is the trend towards more simple refactorings such as basic operations on fields and methods. Figure 2 illustrates this feature by the relatively large number of refactorings towards the right hand side of the figure. More involved refactorings (such as those requiring manipulation of the inheritance hierarchy, e.g., extract subclass, encapsulate downcast and push down method) were not found to occur in large numbers. We thus suggest that developers avoid more involved refactorings, especially those requiring changes to, and manipulation of the inheritance hierarchy. We also suggest that the most common refactorings are those more in-line with typical changes a system may undergo (i.e., field and method operations).

In terms of Fowler's four categories of refactorings, we thus found very little evidence of the 'Dealing with Generalisation' category yet a large number falling into the 'Making Method Calls Simpler' and 'Moving Features Between Objects' categories. Very little evidence of refactorings from the 'Organizing Data' category were evident.

4.2 Supposition Two

The second supposition investigates whether there are any trends within each system across the fifteen refactorings. Since we would expect more refactoring effort to be carried out as a system grows older, our hypothesis would be that refactoring tends to take place towards the later versions of the system rather than earlier in its lifetime. We accept that the systems we looked at are still 'live' and will probably evolve through many more versions before they become obsolete. However, on the basis that we cannot necessarily predict the lifetime of a system, we still feel this supposition to be an interesting one to investigate.

Figures 3, 4 and 5 illustrate data for three of the systems; we have chosen the three systems with the most releases to use as a basis of this analysis (and because of space limitations in this paper). For the Velocity system (Figure 3), relatively few refactorings appear to happen in later versions of the system; equally, relatively few refactorings occur in earlier versions of the software. The bulk of the refactoring activity seems to happen in the mid-versions of the system. We note that the order of the bars for each refactoring follows the order of the comparisons in the legend; Velocity10VS101.xml denotes the XML representation generated by our tool as a result of comparing the two consecutive versions 1.0 and 1.01 of Velocity software.

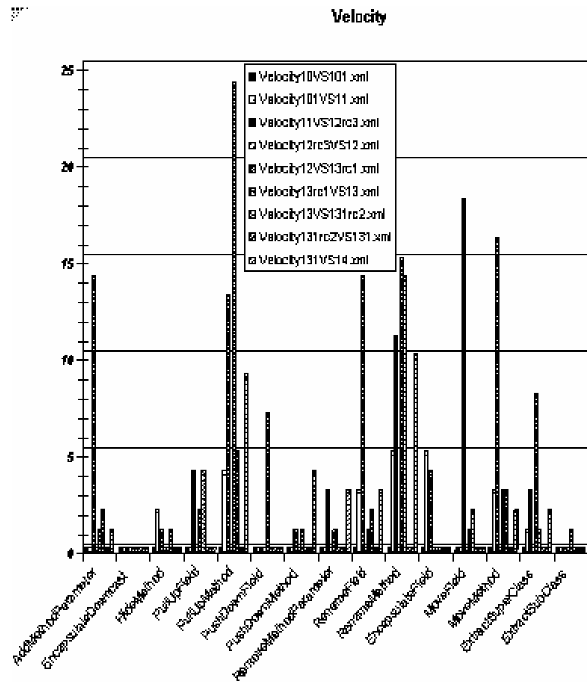

Figure 3: Refactorings for the Velocity system

For the Tyrant system (Figure 4), there is a clear trend of refactorings happening towards later versions of the system (as we hypothesised) in contrast to the Velocity system. It is interesting to note that activity for both the 'Rename Method' and 'Rename Field' features prominently in both systems.

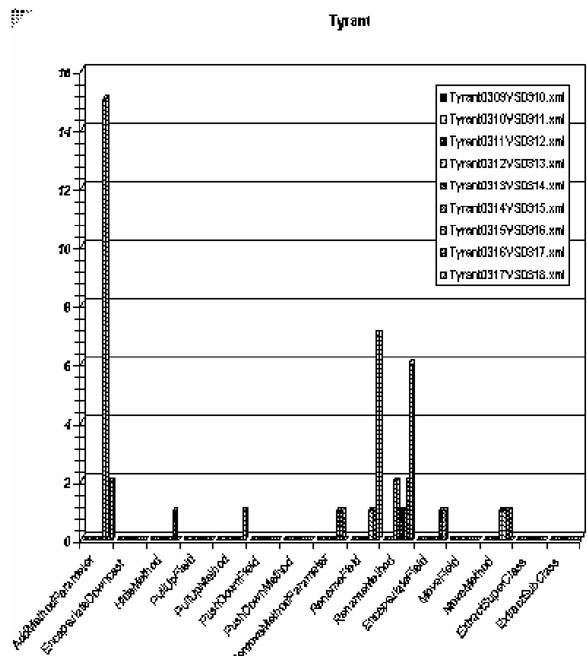

Figure 4: Refactorings for the Tyrant system

For the PDFBox system (Figure 5), the bulk of the refactoring effort seems to occur at both the middle and end of the versions in contrast to the single trends of Velocity and Tyrant. Table 2 summarises for the seven systems the number of refactorings carried out in each version.

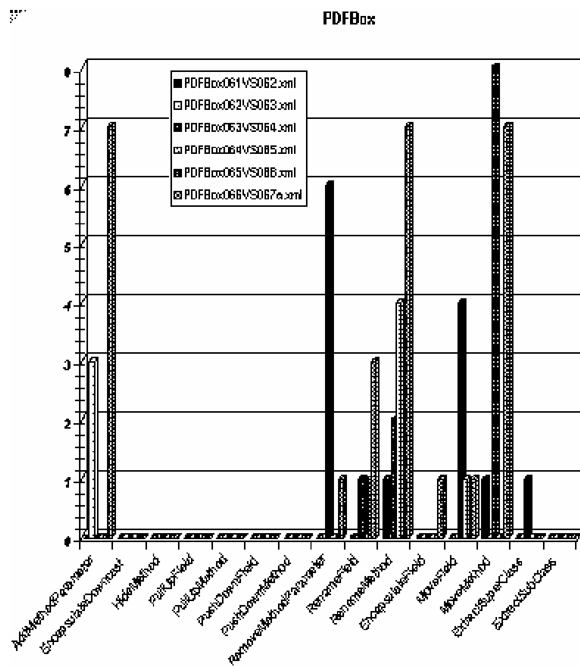

Figure 5: Refactorings for the PDFBox system

Table 2: Summary of refactorings across the versions of the seven systems

Vers	Meg.	Tyr.	Vel.	Antl.	HSQ.	Jaspe.	PDFBox
1-2	1	0	0	4	80	2	2
2-3	0	0	23	37	78	9	3
3-4	0	0	102	1	307	4	22
4-5	0	2	65	0	0	-	5
5-6	0	1	34	-	-	-	1
6-7	0	0	1	-	-	-	27
7-8	0	19	34	-	-	-	-
8-9	-	7	-	-	-	-	-
9-10	-	17	-	-	-	-	-

From Table 2, no clear pattern emerges in terms of when refactorings are carried out. We would therefore conclude that as a system evolves, it is not necessarily the case that increasing amounts of refactoring effort is undertaken. Similarly, it is not the case that large amounts of effort are invested in refactoring effort in the earlier versions of the systems (which is probably more plausible as an hypothesis).

Of the total number of refactorings, the overwhelming majority were carried out in versions 2-3 and 3-4. One suggestion for this trend may be that it takes two or three versions of a system to evolve before the 'decay' starts to creep in. In other words, systems retain a certain stability (in a refactoring sense) for several versions before it becomes worthwhile (and necessary) to undertake any changes

An interesting feature of the refactoring data presented is the tendency for a 'peak' and 'trough' in refactoring effort. For example, for the HSQLDB system, zero changes were made in version 4-5 after 307 refactorings in version 3-4. The same phenomenon is evident in Antlr, PDFBox and to a lesser extent the Velocity and Tyrant systems. This might suggest that after completing a series of refactorings in version *X*, very few refactorings of the type described are needed in version *X+1*. It is worth noting that HSQLDB also saw the highest rise in the number of classes over the versions we investigated; this is reflected in the relatively large numbers of refactorings across the versions of this system.

4.3 Supposition Three

The third supposition investigated was whether any trends in the type of refactoring undertaken across different versions of the systems studied were evident. For example, do certain types of refactoring occur in similar versions. Supposition three also investigated the possibility that refactorings are connected in some sense. The mechanics of all refactorings advise the use of other refactorings [8]. Table 3 shows for each transition between consecutive versions, the total number of each type of refactoring undertaken across the seven systems. We remark that ten is the maximum number of versions (Tyrant); the contribution to Table 3 of systems with fewer than ten versions is thus zero.

One noteworthy feature of Table 3 is the trend of refactoring effort in earlier versions of the systems (in transitions 2-3 and 3-4) and an almost complete absence of refactoring in 4-5. This dip in the refactoring effort supports our 'peak' and 'trough' theory about refactoring. Another feature of this data is that from versions 5 to 9, the trend in refactorings is downward; it then rises sharply. Figure 6 graphically illustrates the trend across versions in terms of total number of refactorings. The general trend however, is for reduced refactoring effort as time evolves

Table 3: Refactorings across the different versions of seven systems[1]

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10
EncD	0	0	0	0	0	0	0	0	0
PudM	0	0	1	0	1	0	0	0	4
ExSub	0	2	3	0	1	0	0	0	0
EncpF	0	5	4	0	0	1	0	1	1
HidM	3	6	2	0	0	1	0	1	0
PupF	0	1	7	0	2	4	0	0	0
ExSup	0	2	10	0	8	1	0	0	2
RemP	3	2	12	0	1	1	1	1	3
PudnF	0	16	3	0	7	0	0	0	0
PupM	0	9	17	0	24	5	0	0	10
AddPa	13	16	41	1	1	9	15	0	3
MovM	14	10	49	0	3	11	1	1	2
MovF	6	45	79	1	1	3	0	0	0
RenM	19	15	71	6	16	21	1	2	16
RenF	31	22	37	0	2	5	1	1	10
Total	89	151	236	8	67	61	17	7	51

The third supposition also investigates the possibility that certain refactorings are connected. In other words, when one refactoring, for example, move method, is performed, there is always (or most times) an accompanying 'move field' refactoring. To understand more fully the relationships between certain refactorings, we begin by describing an accompanying analysis of the relationships between the different refactorings described by Fowler.

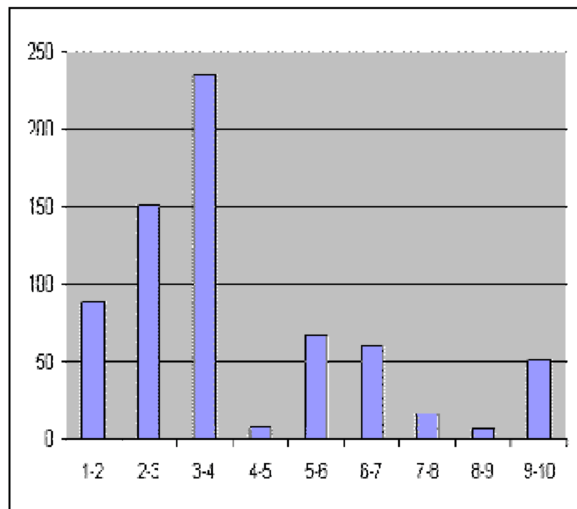

Figure 6: Total refactorings across versions

[1] The first column refers to the refactoring type. Due to place restrictions, full names are replaced by abbreviations. The order of refactorings follows the same of Table 1.

4.3.1 A dependency analysis. Data analysis has shown that the majority of the refactorings fall into two categories of refactoring originally envisaged by Fowler. As part of our ongoing refactoring research, we developed a dependency diagram which showed the inter-relationships between the 72 refactorings. The diagram was developed by hand using Fowler's text as a basis. As a result of producing this graph, it becomes possible to see the likely implications of undertaking a specific refactoring in terms of how many other potential refactorings either **must** be carried out or **may** be carried out at the same time.

For example, for the 'Encapsulate Field' refactoring, Fowler himself suggests that one possible implication of the refactoring is that 'once I've done Encapsulate Field I look for methods that use the new methods' (i..e, accessors needed for the encapsulated field) 'to see whether they fancy packing their bags and moving to the new object with a quick Move Method'.

The encapsulate field refactoring thus has only one possible 'dependency'. From a developer's point of view, the encapsulate field is an attractive and relatively easy refactoring to complete. The 'Add Parameter' refactoring falls into the same category as the Encapsulate Field refactoring. It does not need to use any other refactorings. The only other refactoring that it may consider using is the 'Introduce Parameter Object' refactoring where groups of parameters which naturally go together are replaced by an object.

The extract subclass refactoring, on the other hand, requires the use of six (possible) other refactorings, two of which are mandatory. It has to use 'Push Down Method' and 'Push Down Field' as part of its mechanics. It *may* (under certain conditions) also need to use the 'Rename Method', 'Self Encapsulate Field', 'Replace Constructor with Factory Method' and 'Replace Conditional with Polymorphism' refactorings. The extract superclass refactoring requires a similar number of refactorings to be considered. In fact, for most of the refactorings involving a re-structuring of the inheritance hierarchy, the mechanics are lengthy (requiring many steps and testing along the way).

4.3.2 Connections between refactorings. One explanation for the result in Table 1 (i.e, the high values for simple refactorings and the low values for more 'complex' refactorings) could be attributed to the relative effort required (in terms of activities required) to complete the refactoring. The testing effort of more complex refactorings has also to be considered; the more changes made as part of the

refactoring then *mutatis mutandis*, the more testing would be required.

In terms of whether refactorings are somehow linked, we can see from Table 3 that when the extract superclass refactoring is evident, the pull up method is also a feature. The mechanics of the extract superclass refactoring insist that pull up method is part of that refactoring. Equally, there seems to be evidence of pull up field for the same refactoring (also a part of the extract superclass refactoring). Rename field and method also seem to feature when extract superclass is carried out; rename method (but not rename field) play an important role in the extract superclass refactoring. The rename field refactoring is not specified in Fowler's text. This is interesting since it suggests that may be some effects of refactoring which aren't covered by the refactoring according to Fowler.

Extract subclass also requires use of the rename method refactoring, which may explain the high numbers for that refactoring. To try and explain the high numbers of rename field refactoring, one theory may be that developers automatically change the name of fields when methods are 'pulled up' (in keeping with the corresponding change of method name). A conclusion that we can draw is that there may well be relationships between some of the fifteen refactorings in line with the mechanics specified by Fowler in [8]. However, we suggest that most of the simple refactorings were not as part of any larger refactoring. In the following section, we discuss a number of issues related to our study.

5. Discussion

There are a number of threats to the validity of this study that have to be considered. Firstly, the systems chosen for analysis were open-source systems rather than commercial systems developed and maintained by traditional teams of programmers. In defence of this threat however, we feel that the results described in this study are as valid as any for commercial systems [7]. The results inform our understanding of how open source systems evolve and are maintained. Parallel studies on commercial systems developed in the traditional way would not necessarily detract from these results, but we feel add to them.

The second threat is that we chose seven systems of largely differing application domains; systems of identical application domain may have provided more relevant results. In defence of this criticism, we would claim that for the results described in this paper to be generalised, we would want systems of different application domains.

Another threat might be that we have looked at different changes due to refactoring and ignored the vast number of other types of refactorings and changes which can be applied to software. In terms of other refactorings, the intention of the study was to choose a subset of the seventy-two refactorings which we believed would provide a cross-section of the types of change typically made to software.

One final threat to the validity of the study is fact that we have ignored any consideration of other types of change to the systems investigated which would inevitably have been made. We also need to consider that contrary to the definition of refactoring given in Section 1, refactoring often involves changes to the internal structure of the system which do change its external behaviour. In other words, not all refactorings are semantic preserving. We also have to consider the possibility that some connections between refactorings may exist but cannot be identified unless we widen the number of refactorings investigated. In the next section, we draw some conclusions and point to future work.

6. Conclusions and Future work

In this paper, we have described a study of the refactoring trends across different versions of seven systems. A software tool was used to extract the different refactorings which the software had undergone. Results showed that the majority of refactorings were relatively simple and easy to apply. Those related to structural changes did not seem particularly common. Results also showed that no clear patterns when refactoring was carried out emerged, although a 'peak' and 'trough' effect in terms of refactoring effort was observed. One theory is that perhaps refactoring effort is done in bursts and the system left to settle before further refactoring is attempted. Other results suggest that there are links between complex refactorings and the 'core' (simpler) refactorings which are part of those larger refactorings. Of the large numbers of smaller refactorings we believe that most are carried out independently of any larger refactorings. Finally, and interestingly, it seemed to take two or three versions of a system before any major refactoring effort was observed, suggesting that systems may not start 'decaying' until that point.

In terms of future work, it would be interesting to investigate whether any relationship existed between the refactorings identified and the bugs found across the different releases of the seven systems. The intention of the authors is to replicate a number of recent studies on versions of software and the link with

faults [17]. We also intend extending both the number of refactorings which the tool is capable of extracting and the number of systems. It would also be interesting to run the tool on versions of commercial systems written in a more traditional way (i.e., non open source systems) to see if common features exist.

References

[1] D. Advani, Y. Hassoun and S. Counsell. Heurac: A heuristic-based tool for extracting refactoring data from open-source software versions. Technical Report BBKCS-05-01, Birkbeck College, School of Computer Science and Information Systems, 2005.

[2] D. Arsenovski. Refactoring – elixir of youth for legacy VB code. Available at: www.codeproject.com/vb/net/Refactoring_elixir.asp.

[3] L. Briand, C. Bunse and J. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Transactions on Software Engineering, 27(6), 2001, pages 513—530.

[4] S. Counsell, G. Loizou, R. Najjar, and K. Mannock. On the relationship between encapsulation, inheritance and friends in C++ software. Proceedings of International Conference on Software System Engineering and its Applications (ICSSEA'02), Paris, France, 2002.

[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] 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,

[7] R. Ferenc, I. Siket, T. Gyimothy. Extracting Facts from Open Source Software. Proceedings of 20th International Conference on Software Maintenance (ICSM 2004), Chicago, USA, pages 60-69.

[8] M. Fowler. Refactoring (Improving The Design of Existing Code). Addison Wesley, 1999.

[9] R. Harrison, S. Counsell and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems, Journal of Systems and Software, 52, 2000, pages 173—179.

[10] R. Johnson and B. Foote. Designing Reusable Classes, Journal of Object-Oriented Programming 1(2), pages 22-35. June/July 1988.

[11] J. Kerievsky, Refactoring to Patterns, Industrial Logic, online at: www.industriallogic.com, 2002.

[12] R. Najjar, S. Counsell, G. Loizou and K. Mannock. The role of constructors in the context of refactoring object-oriented software. Seventh European Conference on Software Maintenance and Reengineering (CSMR '03). Benevento, Italy, March 26-28, 2003. pages 111 – 120.

[13] R. Najjar, S. Counsell and G. Loizou. Encapsulation and the vagaries of a simple refactoring: an empirical study. SCSIS-Birkbeck, University of London Technical Report, BBKCS-05-03-02, 2005.

[14] M. O'Cinneide and P. Nixon. Composite Refactorings for Java Programs. Proceedings of the Workshop on Formal Techniques for Java Programs. ECOOP Workshops 1998.

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

[16] B. Foote and W. Opdyke, Life Cycle and Refactoring Patterns that Support Evolution and Reuse. Pattern Languages of Programs (James O. Coplien and Douglas C. Schmidt, editors), Addison-Wesley, May, 1995.

[17] T. J Ostrand, E J. Weyuker and R. M. Bell. Where the bugs are. Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis**,** Boston, Massachusetts, USA. Pages: 86 – 96, 2004.

[18] D. Perry. Laws and Principles of Evolution, Panel Paper, International Conference on Software Maintenance, Montreal, Canada pages 70-71, 2002.

[19] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. Automated Software Engineering, 8:89-120, 2001.