CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Electronic Communications of the EASST (European Association of Software Science and Technology)

Electronic Communications of the EASST
Volume 8 (2008)

Proceedings of the
Third International ERCIM Symposium on
Software Evolution
(Software Evolution 2007)

Package Evolvability and its Relationship with Refactoring

A. Mubarak, S Counsell, R. Hierons and Youssef Hassoun

15 Pages

Guest Editors: Tom Mens, Maja D'Hondt, Kim Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Package Evolvability and its Relationship with Refactoring

A. Mubarak[1], S. Counsell[1], R.M. Hierons[1] and Y. Hassoun[2]

[1] {asma.mubarak, steve.counsell, rob.hierons} @brunel.ac.uk
Department of Information Systems and Computing
Brunel University, Uxbridge, Middlesex, UK

[2] youssef.hassoun@kcl.ac.uk
Department of Computer Science, King's College, London, UK

Abstract

In this paper, we address a set of research questions investigating trends in changes to an open-source system (OSS). An interesting 'peak and trough' effect trend was found to exist in the system studied, suggesting that developer activity comprises of a set of high and low periods. Trends in overall changes applied to the system were complemented with empirical evidence in refactoring data for the same system; this showed a similar peak and trough effect but at different versions of the same system. This result suggests a contrasting motivation between *regular* maintenance practice and that of refactoring. Our analysis of high-level package trends informed some interesting cross-comparisons with refactoring practice, and some insights into why refactoring might be applied after a burst of regular change activity, rather than consistently. We use data extracted from seven Java OSS as a basis for our refactoring analysis.

1. Introduction

A software system is modified and developed many times throughout its lifetime to maintain its effectiveness. In general, it grows and changes to support increases in information technology demands. While from a research perspective, we know a reasonable amount about facets of Object-Oriented (OO) and procedural system evolution Belady and Lehman (1976), Bieman et al., (2003), Arisholm et al., (2006), relatively less well-understood is whether changes at the package level exhibit any specific trends. The benefits of such a study are clear. Understanding changes at higher levels of abstraction may give a project manager a much more *broad-brush* idea of likely, future maintenance or refactoring opportunities. In particular, such a study may also be able to focus developer effort in specific areas of packages susceptible to large numbers of changes. A further area of interest to OO researchers and practitioners is the link between maintenance as part of every system's evolution and that dedicated to refactoring Fowler (1999), Mens and Tourwe (2004). In this paper, we investigate the trends in versions of the 'Velocity' Open-Source System (OSS), with respect to added classes, methods, attributes and lines of code. To support our analysis, we also looked at empirical refactoring data for the same system and associated trends for two other Java OSSs.

We suggest that if the set of *regular* (i.e., essential) maintenance changes exhibit specific characteristics then a set of specific refactorings will also exhibit similar features. Results showed an interesting discrepancy between trends in those regular changes made to the system studied and those as part of a specific set of changes according to refactorings specified in Fowler (1999).

The remainder of the paper is organised as follows. In the next section, we present related work and in Section 3, we provide details of the data analysis addressing three research questions. In Section 4, we tie the trends in package data to those found in a refactoring study using seven Java OSS. Finally, we conclude and point to possible future work in Section 5.

2. Related Work

From an empirical stance, the relationship between OO classes and packages is not well-defined. Ducasse et al. (2005) suggest that it is necessary, for the re-engineering and development of OO systems, to recognize and investigate *both* sets of classes and packages. Focusing on the latter provides a means of comparison with previous work focusing on class changes only. The same authors also suggest that packages have varying functions: they may include utility classes or they may include some fundamental sub-classes enlarging a framework. Ducasse et al. (2004) suggest that the cost of modifying a program may be influenced by the relationship between packages and their enclosed classes. To support the developer in achieving a mental image of an OO system and understanding its packages, they introduce a top-down engineering method based on visualization. Consequently, they raise the abstraction level by detecting packages rather than classes; classifying packages at a high-level prevents developers from being flooded with information.

Bieman et al., (2003) found a relationship between design structures and development and maintenance changes. The same study tried to examine whether potential changes to a class could be predicted by the architectural design context of a class. They found that in four of five case-studies, classes which have function in design patterns were modified more frequently than other classes Gamma et al., (1995). In terms of the architecture of a system, Bieman et al., (2001), found that classes belonging to a design pattern were the most change-prone classes in a system (this might also suggest that change-prone classes are implemented by design patterns). Finally, Demeyer et al. (2000) identified refactoring indicators when comparing different releases of a software system. They used four heuristics to find refactorings; each was identified as a mixture of change metrics.

In this paper, we also consider relationships between trends in changes at the class level with refactoring data extracted using a bespoke tool. In terms of early work in the area, a key refactoring text is that by Fowler (1999). In this text, Fowler describes the mechanics of seventy-two different refactorings in four key categories and describes assorted 'bad smells' in code. According to Fowler, the key indicator of when refactoring is overdue is when code starts to 'smell'. The earlier work of Johnson and Foote (1988) and of Foote and Opdyke (1995) have all made significant contributions to the refactoring discipline and also helped to demonstrate the viability and potential of refactoring. Recent empirical work by Najjar et al.,

(2003) has shown that refactoring can deliver both quantitative and qualitative benefits - the refactoring ‘replacing constructors with factory methods’ of Kerievsky (2004) was used as a basis. The mechanics of the refactoring require a class to have its multiple constructors converted to normal methods, thus eliminating the code ‘bloat’ which tends to occur around constructors. Results showed quantitative benefits in terms of reduced lines of code due to the removal of duplicated assignments in the constructors as well as potential qualitative benefits in terms of improved class comprehension. A full survey of recent refactoring work can be found in Tourwe and Mens (2003).

3. Data Analysis

The main objective of the research described is to assess how a system changes through the analysis of packages in the system and to compare that data with corresponding results from refactoring the same system. Knowledge of trends and changes within packages is a starting point for an understanding of how effective the original design may have been, how susceptible types of packages may be to change and can also inform our knowledge of facets of software such as coupling and cohesion. To this end, a case study approach was adopted using multiple versions of an evolving system. This system was a large OSS called ‘Velocity’ – a template engine allowing web designers to access methods defined in Java. For each version, we collected the number of added classes, lines of code (LOC), methods and attributes. Hereafter, we define a LOC as a single executable statement; we therefore disregard comment lines and white space from calculation of LOC.

3.1 The Three Research Questions

- **RQ1**: Does the number of new classes over the course of nine selected versions increase constantly? This question is based on the notion that a system will grow over time in a constant fashion in response to regular changes in requirements.
- **RQ2**: Is the increase in LOC over the course of the nine versions constant? This question is based on the assumption that the change in LOC over the nine versions will always increase due to evolutionary forces.
- **RQ3**: Is the increase in the number of attributes and methods in a package constant across the versions of a system? This question is based on the assumption that the change in the number of attributes and methods will increase consistently over time in response to constant changes in requirements.

3.1.1. Research Question 1 (RQ1)

Table 1 shows the number of packages in each of the nine versions, the number of new classes across those packages, the number of new classes in total, the maximum increase in classes and the package name where that increase took place. In each of the nine versions, new classes were added to packages and the number added varied significantly from one version to

another. Between versions three and four and six and seven, relatively little change can be seen, while the peak of added classes is reached in the fifth version with 2032 new classes added. Clearly, the addition of classes to this system over the versions investigated is not constant. Interestingly, the version with the highest number of new classes was also accompanied by a drop in the number of packages (from 42 to 36). Equally, some of the largest additions to classes were made after only minor changes to the numbers of packages. Both effects may possibly be due to classes being moved around in the same package and simply renamed.

Table 1. Packages and the new classes over the course of nine versions

Version	No. of packages	No. of new classes	Max inc.	Package name
1st	28	788	176	Editor
2nd	32	1116	207	Java
3rd	38	17	5	Core
4th	42	11	3	Javadoc
5th	36	2032	329	Debuggerjpda
6th	39	45	13	Openide
7th	39	297	92	Core
8th	38	1274	357	Web
9th	39	1386	217	Core

A feature not immediately apparent from the data in Table 1 is the peak and trough effect of this data. A graph was therefore used to present the changes in the number of the new added classes (Figure 1). We suggest that this trend is symptomatic of a burst of developer change activity followed by a period of relative stability and accumulation of new requirements, before another burst of change activity. For RQ1, we conclude that the number of new classes over the course of the nine versions increases at an inconsistent rate, rather than remaining constant. It is not the case that there is constant addition of classes to the Velocity system over the nine versions investigated; RQ1 cannot thus be supported.

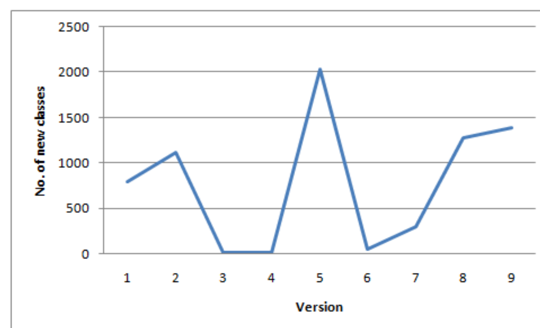


Figure 1. Line chart of new classes added to the packages over the nine versions

3.1.2. Research Question 2 (RQ2)

To investigate RQ2, the 'maximum' increase in the number of LOC among all the versions was used. The data is presented in Table 2. It can be seen that there are increases in LOC over the course of the versions but these increases fluctuate wildly. Interestingly, the Core and Vcscore packages were the packages that saw the maximum increases in LOC for five of the versions. The Core package is the only common package to Tables 1 and 2, suggesting that the addition of a large number of classes does not necessarily imply the addition of a correspondingly large number of LOC. One explanation for this feature might simply be that one class has been split into two (c.f., the 'Extract Class' refactoring of Fowler (1999)).

Table 2. Max. increase in the number of LOC over the course of the nine versions

Version	Max inc in LOC	Package name
1st	3955	Core
2nd	5077	Form
3rd	889	Vcscore
4th	910	Javacvs
5th	6985	Vcscore
6th	1109	Vcsgeneric
7th	369	Core
8th	6418	Core
9th	6743	Schema2beans

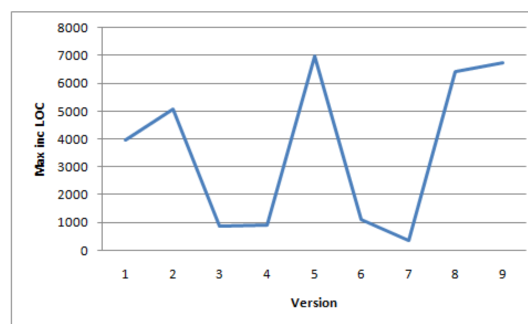


Figure 2. Line chart of the max increase in the number of LOC (nine versions)

Figure 2 confirms that the increases in the number of lines of code over the course of the nine versions fluctuates across versions. Again, the peak and trough effect is apparent from the figure. The most significant changes to Vcscore appear in the first five versions and those of Core appear in the seventh and eighth versions; RQ2 cannot be supported either.

3.1.3. Research Question 3 (RQ3)

For RQ3, the maximum increase in the number of the attributes, and the maximum increase in the number of the methods for each version were used. This data is presented in Table 3 and shows that over the course of the nine versions there are consistent increases in the number of attributes (A) and number of methods (M). However, these increases vary from one version to another. The largest increase in the number of the attributes and methods is at version five. Once again, two packages dominate Table 3 - those being Core and Vcscore (seven of the eighteen entries in columns 4 and 7 relate to these two packages). As per Table 2, the maximum increase in methods occurs at earlier versions for Vcscore, and, for Core, towards later versions.

Table 3. Summary of the increase in the no. of attributes and methods over the nine versions

Version	Inc in A	Max Inc in A	Package Name	Inc in M	Max Inc in M	Package name
1st	153	26	Core	228	36	Vcscvs
2nd	262	49	Form	335	84	Vcscore
3rd	25	7	Jndi	46	11	Vcscore
4th	24	7	Diff	22	6	Diff
5th	325	51	Form	489	70	Vcscore
6th	39	10	Debuggercore	73	14	Openide
7th	17	4	I18n	29	6	Core
8th	238	57	Core	371	150	Core
9th	226	34	Java	378	76	Xml

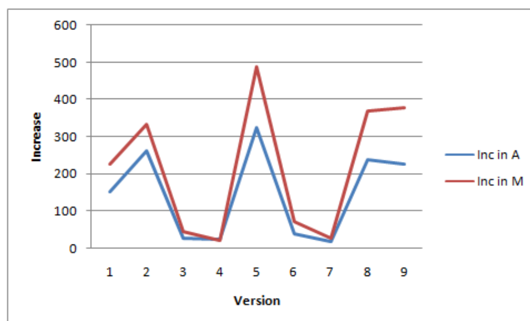


Figure 3. Inc. in attributes and methods

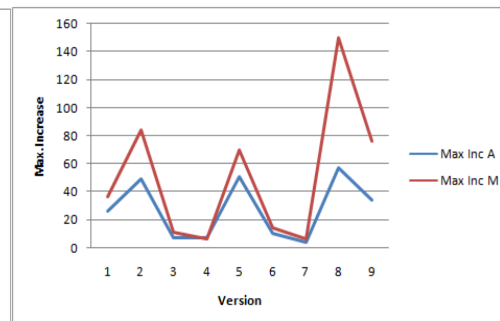


Figure 4. Max. inc. in attributes and methods

Figure 3 shows that the number of the attributes and number of methods both increase during the course of the nine versions, but at a fluctuating rate. Version four shows that more attributes were added than methods; the pattern for all other versions is the opposite. In contrast with the previous analysis, Figure 4 shows that version eight appears to be the source of the largest increase in methods. In keeping with the results from RQ1 and RQ2, we conclude for RQ3 that the increase in attributes and methods is *not* constant across the nine versions investigated. While the results so far give a fairly intuitive understanding of how a system might evolve, what is not so clear is the relationship between the 'regular' set of

changes as we have described them, and the opportunities for undertaking a set of changes such as those associated with refactoring techniques Opdyke (1992), Tourwe and Mens (2003). These are both interesting and potentially fruitful areas of refactoring research as well as challenges facing the refactoring community, Mens and van Deursen (2003).

4. Refactoring Relationships

A current, open research question is whether there is any identifiable relationship between changes as we have described so far and specific refactorings. Beck (2002) suggests that a developer should refactor 'mercilessly' and hence consistently. We would therefore expect refactorings for the Velocity System to be consistently applied across all versions.

4.1 Velocity

Table 4 shows the fifteen refactorings collected by a software tool as part of a full study of refactoring in seven Java OSS systems by Advani et al., (2006). The fifteen refactorings were chosen by two developers with industrial experience and reflected, in their opinion, in consultation with Fowler's text, the common refactorings likely to be made by developers over the course of system's life. As such, refactorings embracing Inheritance, Encapsulation, Movement of Class features and their addition and removal all feature amongst the fifteen refactorings.

Table 4. Refactorings for the Velocity system across nine versions

No.	Refactoring	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6	Ver7	Ver8	Ver9
1.	AddParameter	0	0	**14**	0	**1**	**2**	0	0	1
2.	EncapsulateDowncast	0	0	**0**	0	**0**	**0**	0	0	0
3.	HideMethod	0	2	**1**	0	**0**	**1**	0	0	0
4.	PullUpField	0	0	**4**	0	**2**	**4**	0	0	0
5.	PullUpMethod	0	4	**13**	0	**24**	**5**	0	0	9
6.	PushDownField	0	0	**0**	0	**7**	**0**	0	0	0
7.	PushDownMethod	0	0	**1**	0	**1**	**0**	0	0	4
8.	RemoveParameter	0	0	**3**	0	**1**	**0**	0	0	3
9.	RenameField	0	3	**14**	0	**1**	**2**	0	0	3
10.	RenameMethod	0	5	**11**	0	**15**	**14**	0	0	10
11.	EncapsulateField	0	5	**4**	0	**0**	**0**	0	0	0
12.	MoveField	0	0	**18**	0	**1**	**2**	0	0	0
13.	MoveMethod	0	3	**16**	0	**3**	**3**	0	0	2
14.	ExtractSuperClass	0	1	**3**	0	**8**	**1**	0	0	2
15.	ExtractSubClass	0	0	**0**	0	**1**	**0**	0	0	0

Versions 3, 5 and 6 can be seen as the main points when refactoring effort was applied to the Velocity system (these columns are bolded). In versions 1, 4, 7 and 8, zero refactorings were applied to this system. Figure 5 shows Table 4 in graphical form (with the ‘per version sum’ of the fifteen refactorings on the *y*-axis). The figure shows that refactoring effort is applied most significantly at one version (in this case version 3) and thereafter a peak and trough effect can be seen. Comparing the trend in Figure 5 with that in Figures 1-4 suggests that the majority of the refactoring effort occurred *between* versions where significant changes in classes, LOC, methods and attributes took place; version 3 is a trough in terms of these added features. Conversely, version 5 from Table 4 shows significant refactoring effort to have been applied, coinciding with large changes in the aforementioned features. Version 6 activity (again a trough in terms of Figures 1-4) also shows relatively large amounts of refactoring effort.

A number of conclusions can be drawn from this analysis. Firstly, it is clear that developers do not seem to refactor consistently across the versions of the system studied (Velocity). Secondly, while there is some evidence of peaks in refactoring effort occurring at the *same time* as large changes in classes, LOC, methods and attributes, refactoring seems to occur largely *after* a peak of the same type of changes. One of the claims by Fowler as to why developers do not do refactoring is that they simply do not have the time. Finally, in the preceding analysis, and from Figure 5, it is not the initial versions where the majority of regular change activity is applied to a system. We note, as an aside, that a valuable and interesting analysis of specific dates of version release remains a topic of future work. The first question that naturally arises is why refactoring changes tend to follow the regular changes applied to a system? After all, it is quite feasible for refactoring to be carried out at the same time as other changes (there is limited evidence of this occurring from the data). Moreover, the opportunity for refactoring often arises as part of other maintenance activity and we would thus expect developers to spot opportunities for refactoring as they undertake other work on a system. There is one relatively straightforward explanation for this phenomenon.

All of the fifteen refactorings in Table 4 are semantics-preserving and do not explicitly add large numbers of classes, LOC, methods or attributes as part of their mechanics. For example, the ‘Move Field’ and ‘Move Method’ refactorings would have no net effect on the number of fields or methods in a system, on a package basis. Simple renaming refactorings such as ‘Rename Field’ and ‘Rename Method’ do not, *per se*, add any LOC to the system. Equally, none of the inheritance-related refactorings explicitly add LOC to a system. One further suggestion as to why refactoring occurs at different versions is that after a burst of regular maintenance effort and a new version being released, the decay to the system that those changes have caused may need to be remedied. In other words, after a concerted effort to modify the system through regular maintenance, developers may feel that only *then* is refactoring necessary.

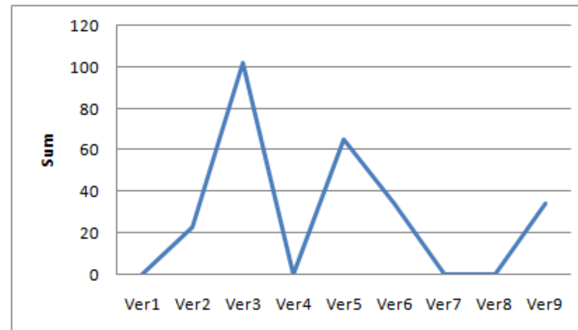
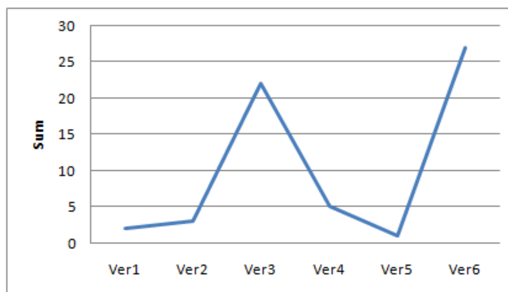


Figure 5. Refactorings in the nine versions of Velocity

This does not explain, however, why for the Velocity system there is significant refactoring effort in version 5 coinciding with a large set of changes in terms of added classes, LOC, methods and attributes? One explanation could be that developers refactor during the course of normal maintenance but without explicitly recognizing it as refactoring. In other words, they may 'tidy up' *in situ*. We could hypothesise that while for Velocity (and the refactorings we have extracted) refactoring effort is not applied consistently, there are two key occasions when, consciously or sub-consciously, it *is* applied.

4.2 PDFBox and Antlr

The question we could then ask is whether refactoring effort is consistent in terms of the versions where it is undertaken and whether a similar trend in refactoring appears in other systems. Figure 6 shows the versions where refactorings were undertaken for the PDFBox system. Versions 3 and 6 appear to be where the majority of the refactoring effort was invested. Although we do not have the dataset of regular maintenance changes applied to the PDFBox system, it is interesting that a peak and trough effect is clearly visible for this system as well as for Velocity.



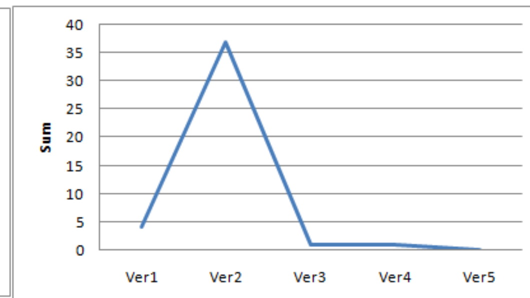


Figure 6. Refactorings for PDFBox

Figure 7. Refactorings for Antlr

Figure 7 shows the refactoring trends for the Antlr system. Version two appears to be the point when most refactoring effort was invested, reinforcing the view that relatively more refactoring seems to be undertaken at early versions of system's life (but not at its inception). It is interesting that across all three systems, version one seems to have been the subject of virtually no refactoring effort. One explanation might be that version one is simply too early in the life of a system for refactoring effort to be applied. On the other hand, it appears that version two or three is when the bulk of refactoring occurs. The question that then arises is whether the numbers of each type of refactorings in each of the three systems were similar? Inspection of the raw data reveals a common trend for refactoring 1 (Add Parameter) and refactorings 9, 10, 12 and 13 (Rename Field, Rename Method, Move Field and Move Method). We hypothesize that these types of refactoring have been applied relatively more frequently than any of the other fifteen because they 'tidy up' a system with relatively little effort being required. After a significant amount of maintenance effort has been applied to a system, minor modifications are bound to be necessary. This may further explain why there is no coincidence between regular maintenance effort and that of refactoring. In the analysis of changes made at the package level, a significant number of methods and attributes were added over the versions studied.

Based on the refactoring evidence, we could claim that the five stated refactorings were a direct response to the problems associated with the addition of so many attributes and methods. For example, the motivation for the 'Move Field' refactoring is when '*a field is, or will be, used by another class more than the class on which it is defined*'. In such a case, the field needs to be moved to the place '*where it is being used most*'. Equally, the 'Move Method' refactoring is applicable when: '*A method is, or will be, using or used by more features of another class than the class on which it is defined*'. For the Velocity system, the large number of these two refactorings at version three suggests that the correspondingly large number of fields and methods added were the cause of required subsequent refactoring. In other words, simple refactorings may have been undertaken to remedy the problems associated with such an intense set of added fields and methods.

We also note that these two refactorings were popular *across all three* systems studied (and at specific points), which adds weight in support of this argument. The same principle applies to simple renaming of fields and methods. It is perfectly reasonable to suggest that when large numbers of attributes and methods have been added to a system, a certain amount of refactoring may be necessary subsequently to disambiguate and clarify the role and meaning of those fields and methods. Fowler (1999) suggests that the 'Move Method' refactoring is the '*bread and butter of refactoring*'. Equally, 'Move Field' is the '*very essence of refactoring*'. Similarly, Fowler reveals an interesting point about the 'Rename Method' refactoring: '*Life being what it is, you won't get your names right the first time*'.

One explanation for the lack of the more 'structurally-based' refactorings (i.e., those that manipulate the inheritance hierarchy) in the systems studied might be that the package access provides the necessary inter-class access that inheritance might otherwise provide. The 'Extract Subclass' and 'Extract Superclass' refactorings would fall into this category. One final point relates to why versions two and three were the source of the most refactoring effort (as opposed to later versions of the system across all three systems). One explanation is that

when a system is at early stages of its lifetime, the design documentation is more likely to be up-to-date. Consequently, the system is relatively easy to modify from a refactoring perspective. As the system ages, increasing amounts of effort and time needs to be devoted to changes as the code 'decays'.

4.3 Across All Systems

One aspect of the analysis that we have not yet considered is the relationship between the refactorings from Table 4. Figure 8 shows the sum of refactorings across all nine versions of the Velocity system (the numerical data for this graph is exactly that in Table 4). Each line in the graph represents the sum of each refactoring for a single version. So, for example, refactoring five (Add Parameter) when taken in totality is a common refactoring across most versions (at least five); the graphs at refactoring 5 show simultaneous peaks. Equally, refactoring ten (Rename Method) can be considered as a popular refactoring in each of the versions. For the fifteen refactorings, a clear trend of peaks and troughs in the fifteen refactorings can be seen. In other words, there is a trend in the propensity of refactorings to occur in 'parallel' (at the same time). Figure 8 thus illustrates the strong bond between the fifteen refactorings.

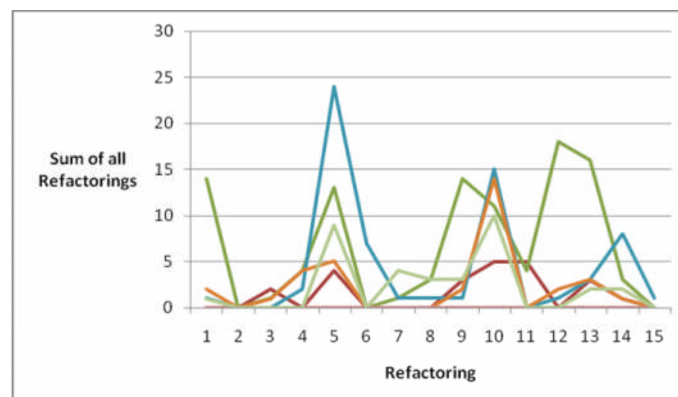


Figure 8. 'Peak and trough' effect of refactorings for Velocity

Two notable exceptions to the trend of refactorings follow peaks and troughs apply to refactoring one (Add Parameter) and twelve (Move Field). At times, there are large numbers of this refactoring in a particular version and very few other refactorings in the same corresponding versions. A simple explanation may account for this trend. They are both refactorings that are used by the mechanics of many other refactorings. They are also two refactorings that a developer may undertake in the course of regular maintenance for example, to fix a fault without the use of any other refactorings. In other words, they can both act as stand-alone refactorings in contexts other than that of refactoring.

4.4 Threats to study validity

There are a number of threats to the validity of the study that need to be considered. One threat to the validity of the study is that we have only considered a relatively small sample of systems to investigate. In defence of this threat, we accept that a larger sample of systems might demonstrate that the results in this paper are more generalisable to the population of systems (external validity). However, the same criticism could be made of a study with double the number of systems studied, for example. The over-arching aim of this paper is to promote further studies and research in this area and that is when the results in this study can be supported or refuted. A further threat to the validity of the study is that we have only considered fifteen refactorings from the 72 stated in Fowler (1999). One plan for future work is to extend the extraction of data to more than fifteen refactorings. We have also only considered a relatively small number of versions of each system; again, in defence of this claim, we chose the most number of versions available at the time the research was being undertaken. One final threat to the validity of the study relates to the time gap between each version of a system. We have assumed, so far, that there is an equivalent time gap between versions and hence that, other things remaining equal, there is a reasonable chance of the same number of refactorings being undertaken between each version. Table 5 shows the time gap in months (m) and days (d) between the nine versions of the Velocity system and the total number of refactorings that were identified in that time - the totals are calculated by summing the individual columns of Table 4. (For the sake of argument, we assume a month to be 30 days duration.) Table 5 shows that there is a wide variation in times between versions of the Velocity system. The minimum gap is 8 days and the maximum gap 8 months, 8 days.

Version	Version gap	Refactorings
Ver 1	-	0
Ver 2	8d	23
Ver 3	10d	102
Ver 4	5m 28d	0
Ver 5	1m 21d	65
Ver 6	6m 28d	34
Ver 7	15d	0
Ver 8	8m 8d	0
Ver 9	7m 9d	34

Table 5. Duration between each version and associated refactorings (Velocity)

What is most interesting and noteworthy from Table 5 is that there is no clear pattern or proportionality with the number of refactorings based purely on the version time gaps. In other words, the length in time between versions seems to have no bearing on the number of refactorings extracted by the tool and undertaken by the developers of this system. For example, the 8 month, 8 day gap between version 7 and version 8 realised zero refactorings. Equally, the 10 days between version 2 and 3 realised the highest number of (102) refactorings. Inspection of the Velocity change logs detailing the changes between versions

revealed a mixture of patches, bug fixes and new requirements. It would therefore seem that refactoring may be motivated by factors other than time *per se*. The amount of developer effort invested into the system between versions, for example, may be a more significant factor than time. A finer-grained analysis of exactly at what date and time the refactorings were undertaken (i.e., a timestamp approach) as well as some indication of effort on the part of the developers might also provide a greater insight and reveal more informative patterns in the refactorings; we leave this detailed aspect of the analysis for future work.

5. Conclusions and Further Work

The goal of this research was to investigate how a system evolved at the package level and this goal was achieved through the use of a case study. A set of three research questions investigated trends in changes of nine versions of an open-source Java system. A bespoke tool was written to extract data relating to changes across those nine versions. An interesting 'peak and trough' effect trend was found to exist in the system studied at specific versions of the system, suggesting that developer activity comprises a set of high and low periods. A contrast was found between those regular changes and those associated with refactoring activity. The results address a hitherto unknown area - that of the relationship between regular changes made to a system as part of maintenance and that of refactoring. While the study describes only a limited sample of systems and evidence of the peak and trough effect is similarly restricted (both threats to study validity), we view the research as a starting point for further replicated studies and for an in-depth and generalised analysis of coupling/refactoring, both inter- and intra-package.

Future work will focus on a number of threads. Firstly, we plan an empirical study to investigate package coupling data across six further OSS using an automated tool called JHawk (2007) to collect relevant metrics. We also plan to use the tool used by Advani et al., (2006) to correlate with the data extracted by JHawk. Finally, we plan on investigating the formal and theoretical properties of refactoring relationships to support the conclusions we have made about the empirical data. For example, to explore the testing implications induced by the empirical relationships described.

Bibliography

Advani, D., Hassoun, Y. and Counsell S. (2006) Extracting refactoring trends from open-source software and a possible solution to the related refactoring conundrum. *Proceedings of the ACM Symposium on Applied Computing,* Dijon, France, April 2006.
Arisholm, E. and Briand, L.C., Predicting Fault-prone Components in a Java Legacy System, *Proceedings of 2006 ACM/IEEE Intl. Symposium on Empirical Software Engineering.*
Beck, K., (1999) Extreme Programming Explained: Embrace Change, Addison Wesley.
Belady, L., Lehman M., (1976) A model of large program development, IBM Sys. Journal, 15(3), pages 225-252.

Bieman, J. M., Straw, G., Wang, H., Munger, P. and Alexander, R. (2003) Design patterns and change proneness: an examination of five evolving systems. In *Proc of the Ninth Int Software Metrics Symposium*, Sydney, Australia, pp.40-49.
Bieman, J. M., Jain, D. and Yang, J. Y. (2001) OO design patterns, design structure, and program changes: an industrial case study. In *Proceedings of the In.l Conf. on Software Maintenance*, Florida, Italy, pp.580-589.
Demeyer, S., Ducasse, S., Nierstrasz, O., (2000) Finding Refactorings via Change Metrics. *In Proceeding of OOPSLA '00: Proc of the 15th ACM SIGPLAN OOPSLA*, Minneapolis, Minnesota, United States. pp. 166–177. ACM Press, New York, NY, USA.
Ducasse, S., Lanza, M. and Ponisio, L. (2005) Butterflies: a visual approach to characterize packages. *In Proceedings of the 11th Intl. Software Metrics Symposium*, Como, Italy, pp.7-16.
Ducasse, S., Lanza, M. and Ponisio, L. (2004) A top-down program comprehension strategy for packages. Technology Report IAM-04-007, University of Berne, Institute of Applied Mathematics and Computer Sciences.
Foote, B., and Opdyke, W., 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.
Fowler, M., (1999) Refactoring (Improving the Design of Existing Code). Addison Wesley.
Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design patterns: elements of reusable object-oriented software.* Massachusetts, Addison- Wesley.
JHawk tool: http://www.virtualmachinery.com/jhawkprod.html, accessed 2007.
Johnson, R., and Foote, B., Designing Reusable Classes, Journal of Object-Oriented Programming 1(2), pages 22-35. June/July 1988.
Kerievsky, J., Refactoring to Patterns, Addison Wesley, 2004.
Mens, T. and van Deursen, A. (2003) Refactoring: Emerging Trends and Open Problems. *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*. University of Waterloo.
Mens, T. and Tourwe, T. (2004) A Survey of Software Refactoring, IEEE Transactions on Software Engineering 30(2): 126-139.
Najjar, R., Counsell, S., Loizou, G., and Mannock, K., The role of constructors in the context of refactoring object-oriented software. *Proc. Seventh European Conference on Software Maintenance and Reengineering (CSMR '03)*. Benevento, Italy, 2003. pages 111 – 120.
Opdyke, W. (1992) Refactoring object-oriented frameworks, Ph.D. Thesis, Univ. of Illinois.
Tourwe, T. and Mens, T., (2003) Identifying Refactoring Opportunities Using Logic Meta Programming, *Proc. 7th European Conf. on Software Maintenance and Re-Engineering*, Benevento, Italy, 2003, pages 91-100.