Extracting Refactoring Trends from Open-source Software and a Possible Solution to the 'Related Refactoring' Conundrum

Deepak Advani and Youssef Hassoun,
School of Computer Science, Birkbeck,
University of London,
Malet Street, London, WC1E 7HX.
{hadva01, yhassoun}@dcs.bbk.ac.uk

Steve Counsell, Department of Information
Systems and Computing,
Brunel University, Uxbridge,
Middlesex. UB8 3PH.
steve.counsell@brunel.ac.uk

ABSTRACT

Refactoring, as a software engineering discipline has emerged over recent years to become an important aspect of maintaining software. Refactoring refers to the restructuring of software according to specific mechanics and principles. In this paper, we describe a tool that allows refactoring data across multiple versions of seven open-source software systems to be collected. The tool automates the identification of refactorings as program transformations between consecutive software releases. The same tool thus allowed an empirical analysis of software development across versions from the perspective of those transformations. We describe results for the systems analysed and point to key conclusions from our analysis. In particular, we investigate a problematic empirical question as to whether certain refactorings are related, i.e., they cannot be undertaken in isolation without other refactorings being undertaken in parallel. In this context, we focus specifically on the four most common refactorings identified by the tool from three of the open-source systems and use a dependency graph to inform conclusions about the empirical data extracted by the tool. An interesting result relating to some common refactorings is described.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Software Metrics; D.2.2 [**Software Engineering**]: Object-oriented design methods.

General Terms

Measurement, Experimentation, Languages.

Keywords

Refactoring, Object-oriented, Tool, Empirical Analysis, Dependencies.

1. INTRODUCTION

One of the key software engineering disciplines to emerge over recent years is that of refactoring [8, 9, 13, 14, 19]. Refactoring can be defined as a change made to software in order to improve its structure (without necessarily changing the program's semantics). The potential benefits of undertaking refactoring include reduced complexity and increased comprehensibility of the code. Improved comprehensibility makes maintenance of that software relatively easy and thus provides both short-term and long-term benefits [21]. Fowler [9] suggests that the process of refactoring is the reversal of software decay and, in this sense that any refactoring effort is worthwhile. Despite the attention that refactoring has recently received, a number of open refactoring issues have yet to be tackled. One of these areas is whether, in an empirical sense, refactorings are related [18], i.e., does one refactoring always incorporate and require the use of one or more other refactorings? The answer to such a question has long-term implications for how we view the time taken to complete and cost associated with, restructuring systems.

In this paper, we describe a tool (and the results from a tool) that extracted refactoring data from seven Java open-source systems. For each release of these systems, the Java source code was parsed and information about class entities collected. Information about fifteen refactorings was saved as an XML format document and then used to compare the different releases. Results indicate the tool to be a useful mechanism for understanding the key type of refactoring undertaken by developers. It also permitted an investigation into the relationship between different refactorings. An interesting result found related to the indirect relationship between certain refactorings - some refactorings were used by many other refactorings but used few themselves.

2. RELATED WORK

In terms of early work in the area, a key refactoring text is that by Fowler [9]. 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 Opdyke [19], of Johnson and Foote [13] and of Foote and Opdyke [8] 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. has shown that refactoring can deliver both quantitative and qualitative benefits [16] - the refactoring 'replacing constructors with factory methods' of Kerievsky [14] 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. Recent work by [1], and on which the research described herein is motivated and based, describes the results of an empirical study of the key trends across multiple versions of the same open source Java software. Relatively few refactorings related to structural change involving an inheritance relationship were identified and surprisingly, no pattern in terms of refactorings across different versions of the software was found. Results thus suggested that developers tend to carry out simple relatively trivial 'core' refactorings at the method and field level, but not as part of larger structural changes to the code (i.e., at the class or hierarchy level). The same research in [1] also identified refactorings according to specific rules and heuristics. In the same vein, developing heuristics for deciding on different refactorings, based on system change data, was earlier investigated by Demeyer et al. [6]. A study of the trends in changes, categorised according to refactorings was also undertaken in [5]. A full survey of recent refactoring work can be found in [15]

In terms of automating the refactoring process, research by Tokuda and Batory [23] 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 manual) refactoring of fourteen thousand lines of program code. Finally, the principles of refactoring are not only limited to object-oriented languages [2]. A major feature of our analysis is the use of metrics to quantitatively capture the features of the system being investigated. Many metrics have been proposed and used for analysing object-oriented and procedural-based software, both theoretically and empirically [3, 4, 10]. Finally, the importance of extracting facts from open-source software, as we have herein, is expressed in [7].

3. THE REFACTORING TOOL

The underlying model used by the tool is XML, together with a set of heuristics to guide the search for each refactoring. The set of values of the entire program are represented as an XML tree consisting of sequences of (sub)-trees representing the individual types. The tool was applied to seven different open-source Java software systems of industrial strength chosen on a random basis to cover a range of application domains. The systems included computer games, a template engine, a compiler construction tool, an SQL database, documentation support and PDF file manipulation. We applied the tool and followed the growth of each of the following systems through their different releases:

1. MegaMek. A computer game. The number of classes and interfaces in this system remained static at 190 and 13, respectively.
2. Tyrant. A graphical-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 has 171 classes and 31 interfaces.
5. HSQLDB. A relational database application supporting SQL. HSQLDB started with 52 classes and 1 interface. The latest version has 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.

For space purposes, we will focus primarily on a random sample of three of these systems (i.e., HSQLDB, JasperReports and Antlr), but note that the same trends found in those three systems were also found in three of the other four systems. At present, the tool covers fifteen different refactorings ranging from simple (i.e., easy to implement) refactorings to more complex refactorings (requiring major changes to the system). The refactorings chosen reflected what we believed to be those most commonly employed by developers:

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. Pull Up Field. 'Two subclasses have the same field'. In this case, the field in question should be moved to the superclass.
5. Pull Up Method. 'You have methods with identical results on subclasses'. In this case, the methods should be moved to the superclass.
6. Push Down Field. 'A field is used only by some subclasses'. The field is moved to those subclasses.
7. Push Down Method. 'Behaviour on a superclass is relevant only for some of its subclasses'. The method is moved to those subclasses.
8. Remove Parameter (from the signature of a method).
9. Rename Field. A field is renamed to make its purpose more obvious.
10. Rename Method. A method is renamed to make its purpose more obvious.
11. Encapsulate Field. The declaration of a field is changed from public to private.
12. Move Field. 'A field is, or will be, used by another class more than the class on which it is defined'.
13. 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'.
14. Extract Superclass. 'You have two classes with similar features'. In this case, create a superclass and move the common features to the superclass.
15. 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.

We remark that some of these refactorings are easier to implement by the tool than others. They are also easier to implement by the developer. For example, renaming and hiding of fields and/or methods are relatively simple refactorings, do not require major program changes and can be identified by our tool relatively easily. Other refactorings, such as extracting sub- and/or superclasses are more complex, require structural changes involving the class hierarchy and are more involved in terms of how they were implemented by the tool; they would also be more complicated refactorings for a developer to implement.

3.1 Refactoring detection criteria

Each refactoring transformation is defined through a set of rules or criteria. For example, to detect whether the 'move field' refactoring has taken place in the transition from one release to the next, the tool checks whether:

1. A field (name, type) appears in a class type (belonging to older version) but appears to be missing i.e., has been dropped from the corresponding type (belonging to later version).
2. The field (name, type) does not appear in any superclass or subclass of the original type.
3. A similar field (name, type) appears to have been added to another type (belonging to a later version) whose corresponding type in a former version, if there is one, does not contain the field in question.

If all criteria (clauses 1 to 3) are met for a field under investigation, the tool reports each such field as an occurrence of that refactoring. The criteria for the 'move method' refactoring are similar – the only difference being the extra information: a method representation includes access label, name, return type and parameter list. The criteria for the 'extract superclass' refactoring are as follows:

1. A class type whose unaccounted fields or methods are pulled up into a newly created superclass (that does not exist in a former release) becomes an extracted superclass.
2. The class from which this superclass was extracted becomes the base type.

The reported XML file includes the base class name with all the fields and methods that have been pulled up to define the new superclass in the later version. The criteria for the 'extract subclass' refactoring is similar to that of the 'extract superclass' refactoring except that instead of pulling the fields or methods up the hierarchy, they are pushed down the hierarchy. For completeness, a broad cross-section of example refactorings identified by the tool were verified by hand.

4. DATA ANALYSIS

Figure 1 shows results in terms of refactorings extracted by the tool from the HSQLDB system (incorporating four versions). The order of refactorings from left to right on Figure 1 (and on subsequent figures) is the same as that of the numbered list 1-15 in Section 3. Figure 1 shows a clear majority of refactorings to fall into the rename method, rename field, move method and move field categories. There is some evidence of 'push down field' and 'add parameter' refactorings. Notable is the relatively small number of these and other inheritance-based refactorings (2-7).

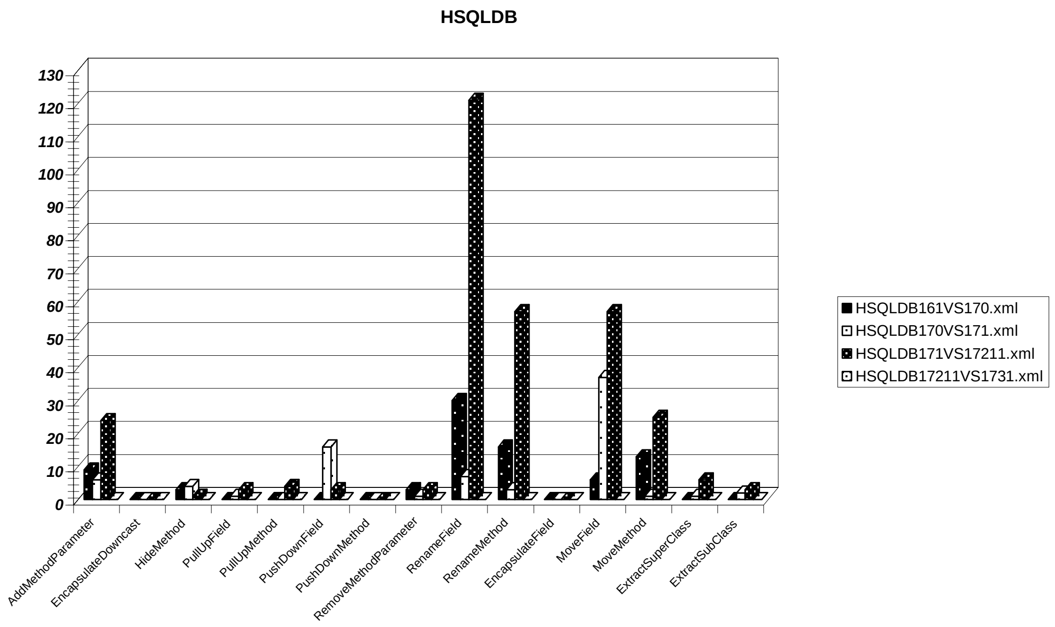

Figure 1. HSQLDB System results.

Table 1 tabulates the frequencies of refactorings that occurred when comparing consecutive releases of the HSQLDB software. The second column refers to refactorings identified in the transition from version one to two; subsequent columns reflect the changes across subsequent versions. Reinforcing the results from Figure 1, we note the high frequency of simple refactorings such as ‘rename method’ and ‘rename field’ compared with relatively lower occurrences of more complex refactorings such as extract subclass and extract superclass. Interestingly, refactorings related to encapsulation of field and objects (encapsulate downcast and encapsulate field) do not figure at all and are zero-valued. The trend for three of the four refactorings (4-7) for HSQLDB is also evident for the JasperReports system shown in Figure 2 (incorporating three versions). It also follows the trend in the HSQLDB system in terms of inheritance-based refactorings. Figure 3 for the Antlr System (incorporating five versions) shows a similar pattern but also shows certain numbers of the pull up method refactoring (where methods with identical results on subclasses are ‘pulled up’). It also has very few inheritance-based refactorings.

Table 1. Refactorings identified between consecutive releases of HSQLDB.

Refac. No.	HSQLDB	Vers. 1 – 2	2 – 3	3 – 4
1	Encapsulate Downcast	0	0	0
2	Encapsulate Field	0	0	0
3	Hide Method	3	4	1
4	Rename Method	16	3	57
5	Rename Field	30	7	121
6	Move Method	13	1	25
7	Move Field	6	37	57
8	Push Down Method	0	0	0
9	Push Down Field	0	16	3
10	Pull Up Method	0	0	4
11	Pull Up Field	0	1	3
12	Add Parameter	9	6	24
13	Remove Parameter	3	1	3
14	Extract Superclass	0	1	6
15	Extract Subclass	0	2	3

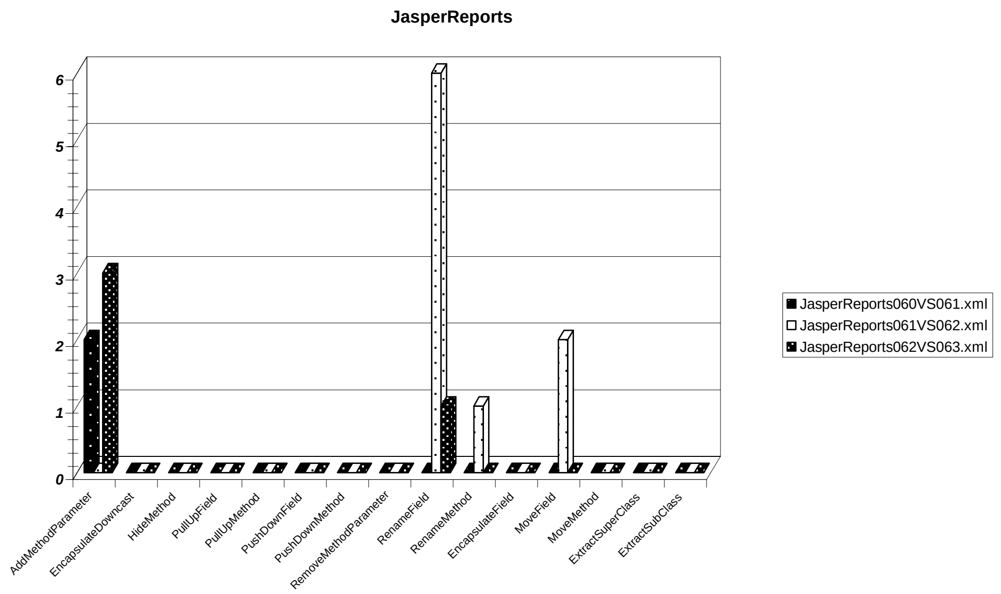

Figure 2. JasperReports System results.

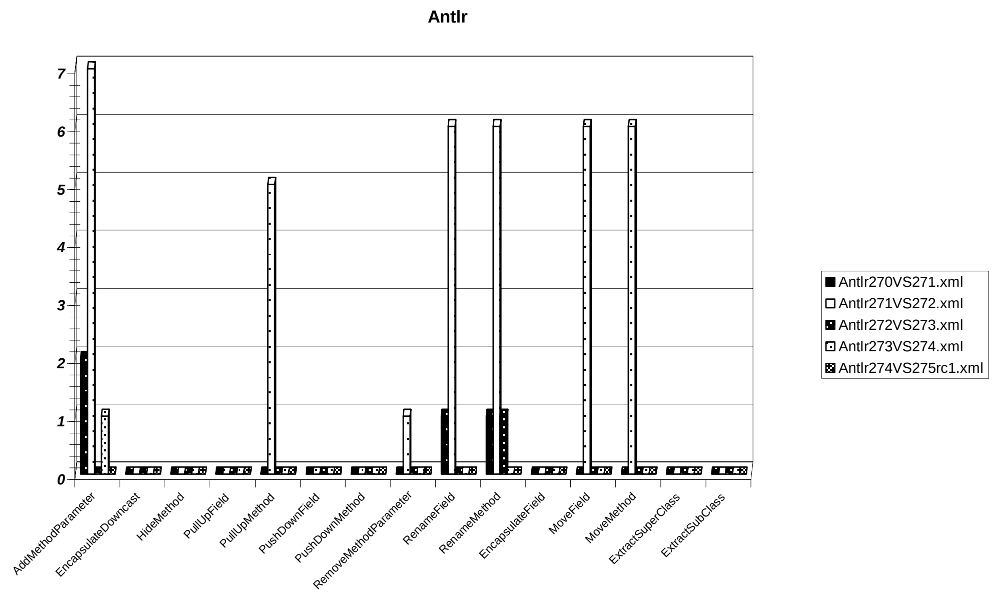

Figure 3. Antlr System results.

For the total number of refactorings identified across the seven systems, four refactorings, namely move method move field, rename method and rename field accounted for approximately 66% of total refactorings identified. A similar pattern to that for the three systems described was found to exist for three of the remaining four systems (MegaMek being the odd system in this case showing no clear patterns). The results were therefore consistent across all but one of the seven systems investigated. Table 2 summarises, for all systems, the number of each of these four refactorings. Interestingly, it would appear that the interval between versions 2 and 4 is when the majority of these four refactorings were undertaken.

Table 2. The four refactorings across all versions (all seven systems).

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10
Move Method	14	10	49	0	3	11	1	1	2
Move Field	6	45	79	1	1	3	0	0	0
Rename Method	19	15	71	6	16	21	1	2	16
Rename Field	31	22	37	0	2	5	1	1	10
Total	70	92	336	7	22	40	3	4	28

The data from Table 2 would seem to suggest a relationship between the four refactorings. In other words, relatively large numbers of these four refactorings are carried out in each version. This would seem to make sense since when we move a method, we also tend to move a field. Equally, when we move a method (or field), we might rename that method (or field). In the next section, we investigate this potential relationship in more detail.

4.1 Relationship between refactorings

Table 3 shows the relationships between the four common refactorings identified by the tool. The table indicates whether for each of the four refactorings, there is a reference to any of other three refactorings in the mechanics specified by and according to Fowler [9]. We note that the 'rename field' refactoring is not part of any refactoring specified in Fowler, but would have similar semantics to the rename method refactoring. It is included to complement the rename method refactoring and inform our overall understanding. We also note that we would expect the data from this empirical study to support the mechanics of each refactoring stated in [9]. In other words, if refactoring A is said to use refactoring B as part of its mechanics, then this is the relationship we anticipate existing in the empirical data. An 'X' symbol entry in Table 3 indicates that for any two refactorings, there is *no* direct reference relationship. For example, the mechanics of the move method refactoring refer to none of the other three refactorings. Equally, there is no relationship between when a method is renamed and when a field is renamed. In fact, Table 3 shows no relationship between any of the four refactorings. This was an interesting and surprising result to emerge from our study and contradicted the authors' intuitive opinion about the behaviour of systems. It would seem at first inspection that the four refactorings act independently of each other.

Table 3. Theoretical relationship between the four refactorings.

Refactoring	Move Method	Move Field	Rename Method	Rename Field
Move Method	-	X	X	X
Move Field	X	-	X	X
Rename Method	X	X	-	X
Rename Field	X	X	X	-

4.1.1 Towards an explanation

According to Fowler, the 'move method' refactoring specifies no other refactoring in its mechanics. The 'move field' refactoring suggests that in certain cases, the 'extract class' and 'encapsulate field' refactorings may play a part. Similarly, the rename method refactoring may require the use of either or both 'remove parameter' and 'add parameter' refactorings. Table 4 shows summaries of results for three of these refactorings across the versions of the seven systems (we remark that the tool does not include the extract class refactoring). We also note that for the three systems we have looked at, only versions 1-4 are relevant for the HSQLDB system and versions 1-3 for the JasperReports system. Equally versions 1-6 only are relevant for the Antlr System. For completeness however, we include all versions for all seven systems of which the maximum was 10 for the Tyrant system.

Table 4. Summary results for three refactorings.

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10
Encapsulate Field	0	5	4	0	0	1	0	1	1
Remove Parameter	3	2	12	0	1	1	1	1	3
Add Parameter	13	16	41	1	1	9	15	0	3

From Table 4, there seems to be evidence of removal and addition of parameters in the same versions noted from Table 2, supporting the view that the values in the two tables may be linked. Although we cannot say for certain that there is any connection between refactorings, it would make perfect sense for a parameter to be added or removed when a method was moved or renamed to reflect new or modified use of the method. Furthermore, the motivation for the move method refactoring according to Fowler supports this view: '*A method is, or will be, using or used by more features of another class than the class on which it is defined*'. Two questions thus emerge from this analysis. Firstly, what is the interplay between the four refactorings described in Table 2? Secondly, what relationships do refactorings such as those described in Table 4 hold with these four refactorings?

4.1.2 A dependency graph

As part of our refactoring analysis, we developed a directed dependency graph showing all seventy-two refactorings and how they were inter-related. In the graph, nodes represented the refactorings and arrows connecting the nodes represented the relationships between those refactorings given by the mechanics of each refactoring in [9]. The size of the graph precludes its inclusion in this paper; however, for each refactoring let's say, X, the in-degree and out-degree taken from the graph illustrated the refactorings that *used* X, and that in turn were *used by* X. Three key characteristics of the dependency graph relevant to the work herein were identified as:

1. Among the refactorings with the highest number of in-degrees (arrows entering the node) were 'Move Method', 'Move Field' and Rename Method'. In other words, these three refactorings were used extensively by other refactorings. The 'Rename Method' refactoring had an in-degree of 9 and an out-degree of zero. The 'Move Method' refactoring had an in-degree of 8 and an out-degree of zero. The 'Move Field' refactoring had an out-degree of two and an in-degree of two. These values contrast strongly with the median in-degree of one and out-degree of one for the remaining sixty-nine refactorings. Many refactorings had zero in-degree and out-degree reflecting the fact that they neither used, nor were used by, any other refactoring.
2. The same three refactorings were not directly linked but were *indirectly* linked. For example, the 'Move Field' refactoring prescribes the use of 'Encapsulate Field' which itself then uses 'Move Method'. The 'Move Method' refactoring is used by the 'Inline Class' refactoring, which then may also use the 'Move Field' refactoring. Equally, the 'Extract Class' refactoring uses both 'Move Method' and 'Move Field'. The 'Encapsulate Collection' refactoring uses both 'Rename Method' and 'Move Method'. A wide range of such relationships were found on the diagram involving the three refactorings. From a visual perspective, the various other refactorings seemed to 'cluster' around the four key refactorings identified.
3. Contrary to that suggested in the previous section, the 'Add Parameter' and 'Remove Parameter' refactorings had no obvious connection with any of the refactorings from Table 2. However, they did appear to cluster around other refactorings in the same way that certain classes clustered around the four refactorings from Table 2. This clustering effect was a counter-intuitive finding of the research.

The data in Table 3 is thus misleading since it gives the impression that there is no connection between any of the refactorings considered in the same table. Given the dependency diagram, we claim that when one of the first three refactorings in Table 3 is used, it is possible that at least one of the other two is used at the same time but indirectly through a different refactoring. We thus conclude that the high numbers of the three refactorings found in Table 2 and for the other systems also are due to the high indirect inter-dependence of the three refactorings by many other refactorings (many of which are not included in the fifteen considered herein). Knowledge of such dependencies between refactorings is a useful tool in the armoury of both the developer and project manager when deciding amongst competing restructuring tasks, allocation of developer time and allocation of cost for highest benefit. Furthermore, knowledge of the clustering of refactorings may have implications for the extent of testing effort required after completing refactorings.

5. CONCLUSIONS

The aim of refactoring transformations is to improve software comprehension and to reverse the effect of software ageing. This paper has presented a tool for automating the analysis of software systems development from a refactoring perspective. Seven open-source Java systems were analysed across a number of versions and refactoring data extracted using a tool. Results showed a tendency towards simple refactorings notably: rename method (and field) and move method (and field). We analysed the results for three of the seven systems and focused on four of the most popular refactorings discovered. The research demonstrated that developers do undertake large numbers of these relatively simple refactorings and there is often an indirect rather than direct relationship between the most common refactorings identified by our tool. It would seem that these four refactorings are 'worker' refactorings providing a service to the other (perhaps more complex) refactorings. The tool and equally importantly the dependency diagram have helped to uncover our understanding of how these refactorings may be related. At present, the underlying model and tool covers only a limited number of refactorings and only tentative conclusions can be drawn. As future work, we are planning to extend to the scope of the tool to cover more refactorings. We are also interested in wrapping the tool in a GUI interface to make the investigation of refactoring trends in software development easier. A further strand of research will be to investigate the relationship between the refactorings described and faults observed for the same systems over the same versions. This research will help to support or refute findings of earlier work in this area [20]. Finally, the dependency diagram needs to be analysed rigorously for mathematical properties to support further analyses of the refactoring data.

6. REFERENCES

[1] D. Advani, Y. Hassoun and S. Counsell. Refactoring trends across N versions of N Java open source systems: an empirical study. SCSIS-Birkbeck, University of London Technical Report, BBKCS-05-03-02, 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 Trans. 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 Intnl. Conf. on Software System Eng. 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 Intnl. Conf. 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 Prog. Systems Languages and Applns (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] 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.

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

[10] 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.

[11] JavaCC, JavaCC Parser, Available [ONLINE] from [https://javacc.dev.java.net/], Last Accessed 22nd August 2004.

[12] JDOM, JDOM API, Available [ONLINE] from [http://www.jdom.org], Last Accessed on 23rd August 2004.

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

[14] J. Kerievsky, Refactoring to Patterns, Addison Wesley, 2004.

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

[16] 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.

[17] 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.

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

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

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

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

[22] Recoder, Recoder API, Available [ONLINE] from [http://recoder.sourceforge.net/], Last Accessed on 23rd August 2004.

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