International Journal of General Systems, Vol. 35, No. 3, June 2006, 329–346

Taylor & Francis
Taylor & Francis Group

Understanding the complexity of refactoring in software systems: a tool-based approach

DEEPAK ADVANI†, YOUSSEF HASSOUN† and STEVE COUNSELL‡*

†School of Computer Science, Birkbeck, University of London, Malet StreetLondon, WC1E 7HX, UK
‡School of Information Systems, Computing and Mathenmatics, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK

(Received 30 July 2005; in final form 28 November 2005)

The majority of information systems these days engender a high level of complexity through the extent of possible inputs to testing, required processing and consequent outputs. In fact, complexity permeates every level of this model for an information system. Complexity thus has a direct effect on the extent to which a system needs to be tested, through those inputs. Complexity also inhibits the ease with which a system can be modified since more time needs to be devoted to assessment of change complexity and resulting tests. Reduction of complexity is the goal of every developer when initially developing a system and, as importantly, after the system has been developed and inevitable changes are made. In this paper, we analyse an automated technique for extracting the typical changes (or refactorings as we have labelled them) made to various Java systems over different versions of its lifetime. Our goal is to identify areas of change where complexity can be examined more thoroughly and aid thus given to the developer when maintaining systems. A generic tool was developed specifically for this task and the results showed new and promising insights into the way systems behave as they evolve. In particular, the complex refactorings are relatively rare compared with more simple refactorings.

Keywords: Software systems; Refactoring; Tool-based approach; Information systems

1. Introduction

One of the key software engineering disciplines to emerge over recent years is that of refactoring (Opdyke 1992, Fowler 1999, Tokuda and Batory 2001). 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 and proposes 72 refactorings in his text (Fowler 1999). 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 (and in theory becomes more complex) 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

*Corresponding author. Email: steve.counsell@brunel.ac.uk

International Journal of General Systems
ISSN 0308-1079 print/ISSN 1563-5104 online © 2006 Taylor & Francis
http://www.tandf.co.uk/journals
DOI: 10.1080/03081070600661051

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 15 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 such as renaming fields and methods, moving fields etc. Very few refactorings related to structural change involving an inheritance relationship were found such as modification of the inheritance hierarchy (e.g. pulling up fields and methods, extracting subclasses and superclasses). 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.

The research addresses several problems currently faced by the software engineering community. Firstly, what is complexity when considering refactoring and trends in refactoring across systems? Secondly, where is refactoring effort being made by developers. Is this a good use of their time? Thirdly, the results can inform future decisions on how to allocate developer testing effort by project managers. For example, what is the quantifiable trade-off between complex and simple refactorings? Finally, what is the relationship between simple and complex refactorings? Such questions may also have implications for the amount of time given over to code optimisation and perfection. The research therefore informs the development process as a whole and seeks to understand in more detail the "complexity of refactoring change" applied to software over its lifetime.

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. 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 (Counsell *et al.* 2003). In that paper, the "substitute algorithm" refactoring (Fowler 1999) (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 analysis of the 72 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. In other words, we would expect a period of high refactoring activity followed by a period of relatively low amounts of refactoring effort. 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 (2002).

In terms of related work, the seminal text and from which our 15 refactorings were taken is that of Fowler (1999). The work of Opdyke (1992), and Johnson and Foote (1988), has also been instrumental in promoting refactoring. Earlier work by Najjar *et al.* (2003) has shown the quantitative and qualitative benefits of refactoring, the refactoring "replacing constructors with factory methods" of Kerievsky (2002), 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.* (2000).

In terms of automating the search for refactoring trends, research by Tokuda and Batory (2001), have 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 14,000 lines of code. Work by Tahvildari and Kontogiannis (2003), investigated the potential for tool use in the identification of opportunities for program transformation. Refactoring is also strongly linked with the Extreme Programming (XP) community; work by Beck (2004), in particular has suggested a strong link between characteristics of the rapid development approach of XP and the need for frequent refactoring changes. A comprehensive survey of the refactoring area can be found in (Mens and Tourwe 2004). Finally, the principles of refactoring are not limited to object-oriented languages; other languages have also been the subject of refactoring effort (Arsenovski).

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 (Harrison *et al.* 2000, Briand *et al.* 2001), thus supporting the view that inheritance-based refactorings are avoided by developers. Finally, very little research has been carried out into composite refactorings (O'Cinneide and Nixon 1998) where one refactoring is followed by *n* other refactorings.

3. Study details

3.1. The fifteen refactorings chosen

The choice of which 15 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 (Counsell *et al.* 2003). 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 nine (rename field) were taken from Fowler's text. We felt that the "Rename Field" was of sufficient interest and had sufficient relationship to the other fourteen refactorings that it should be included in our analysis. The 15 refactorings chosen and the circumstances motivating that refactoring (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 four 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 and 5.
2. *Organising data*: Refactoring 6.
3. Moving features among object: Refactorings 7 and 8.
4. *Dealing with generalisation*: Refactorings 10, 11, 12, 13, 14 and 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 hundreds of classes in each of *n* versions such as those described herein, 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 (Advani *et al.* 2005), 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.

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 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.

We also note that no information relating to which developers had undertaken which changes to versions of the system was included in our analysis. We would expect developers working on open-source systems to be relatively experienced and competent, however. Such an analysis would be the subject of future work.

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. It uses a set of heuristics for each refactoring (described in Section 3.1) to extract the refactoring data.

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. 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 checked 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 1–3) are met for a field under investigation, the tool reported 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.

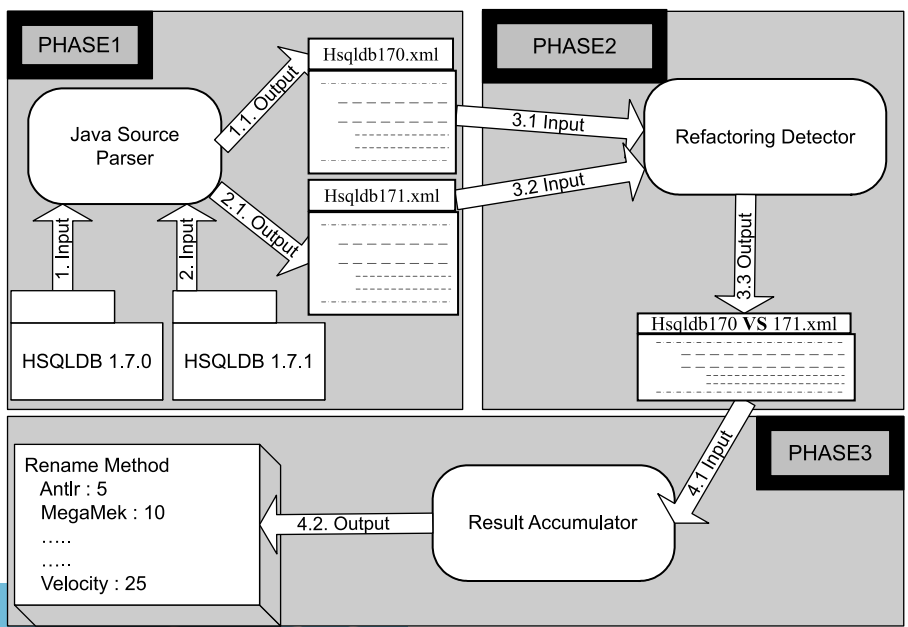

Figure 1. Structure of the refactoring tool.

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. A more detailed description of our tool is provided in (Advani *et al.* 2005).

4. Data analysis

We investigate three suppositions in our analysis of the data. Firstly, we investigate 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. 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 scatter chart. The order of the 15 points is the same as that of table 1.

Figure 2 shows the most popular refactoring to be the "Rename Field" refactoring (point 15), followed by "Rename Method" (point 14). In the HSQLDB system, 158 occurrences of the "Rename Field" refactoring were found out of a total 209. 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 six occurrences, all for the Velocity system). The "Pull Up Method" refactoring value for Velocity is noteworthy

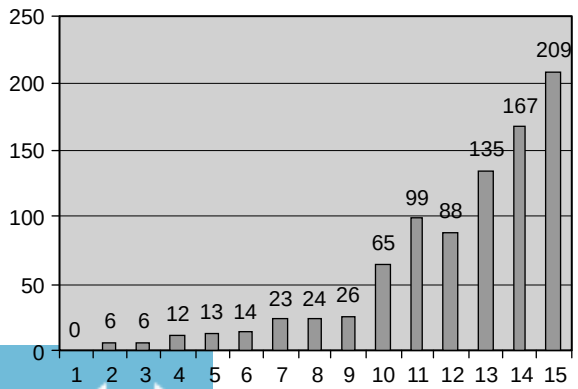

Figure 2. Total refactorings extracted from the seven systems.

since 55 occurrences of this refactoring were found out of a total 65 (we investigate this feature of Velocity in Section 4.2).

One surprising result from figure 2 is the lack of "Encapsulate Field" and "Hide Method" refactorings (points 4 and 5). These are quite simple refactorings; in each case, the mechanics of doing each of these refactorings are simple. 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 have 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 (Counsell *et al.* 2002). Secondly, recent work by Najjar *et al.* (2005), 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 simple.

Although we would not have expected a high number of inheritance-based refactorings, the relatively low values found for the "Encapsulate Downcast", "Push Down Method" and "Extract Subclass" refactorings were surprising. We accept that other inheritance-based refactorings did figure relatively prominently, in particular the "Extract Superclass", "Push Down Field" and "Pull Up Method" refactorings. It seems that creating a superclass is a more popular activity than creation of a subclass.

Perhaps it is the nature of open-source software (where independent developers can make unilateral changes) that explains why changes requiring developers to reorganise the system's structure in the former case do not take place. More studies would be needed before any conclusion could be drawn on this issue, however, table 1 shows the aggregate refactorings for the seven systems together with the maximum and mean value for each refactoring over the seven systems. For example, for the "Push Down Method" refactoring, the numbers of this refactoring that any system observed across its versions totalled six from a total in all seven systems of six. Equally, the numbers of the "Push Down Field" refactoring

Table 1. Refactoring summary data.

No.	Refactoring type	Maximum	Mean	Total
1.	Encapsulate downcast	0	n/a	0
2.	Push down method	6	0.86	6
3.	Extract subclass	5	0.86	6
4.	Encapsulate field	9	1.71	12
5.	Hide method	8	1.86	13
6.	Pull up field	10	2.00	14
7.	Extract superclass	15	3.29	23
8.	Remove parameter	7	3.43	24
9.	Push down field	19	3.71	26
10.	Pull up method	55	9.29	65
11.	Add parameter	39	14.14	99
12.	Move method	39	13.00	88
13.	Move field	100	19.29	135
14.	Rename method	76	23.86	167
15.	Rename field	158	29.86	209

observed by any system across its versions was 19 from a total in all seven systems of 26. The mean represents this total divided by the number of systems (i.e. seven).

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). The "renaming" type of refactorings do not change the underlying abstract syntax tree (AST) representation of the system and in that sense could be claimed to be less "damaging"; this may further explain why they were so relatively popular.

In terms of Fowler's four categories of refactorings, we 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 was evident.

4.2. Supposition two

The second supposition investigates whether there are any trends within each system across the 15 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). 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. The order of the bars for each refactoring follows the order of versions in the legend; version Velocity10VS101.xml denotes the XML representation generated by our tool for two consecutive versions 1.0 and 1.01 of that 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.

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.

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

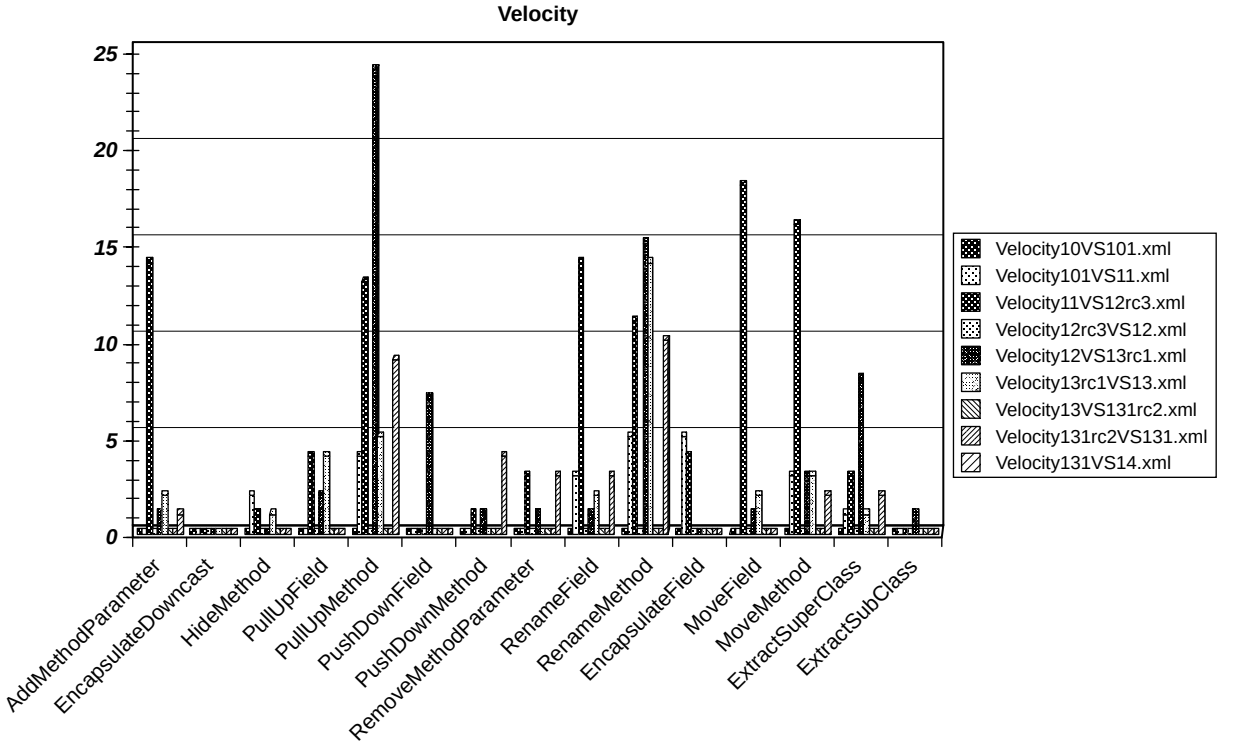

Figure 3. Refactorings for the velocity system.

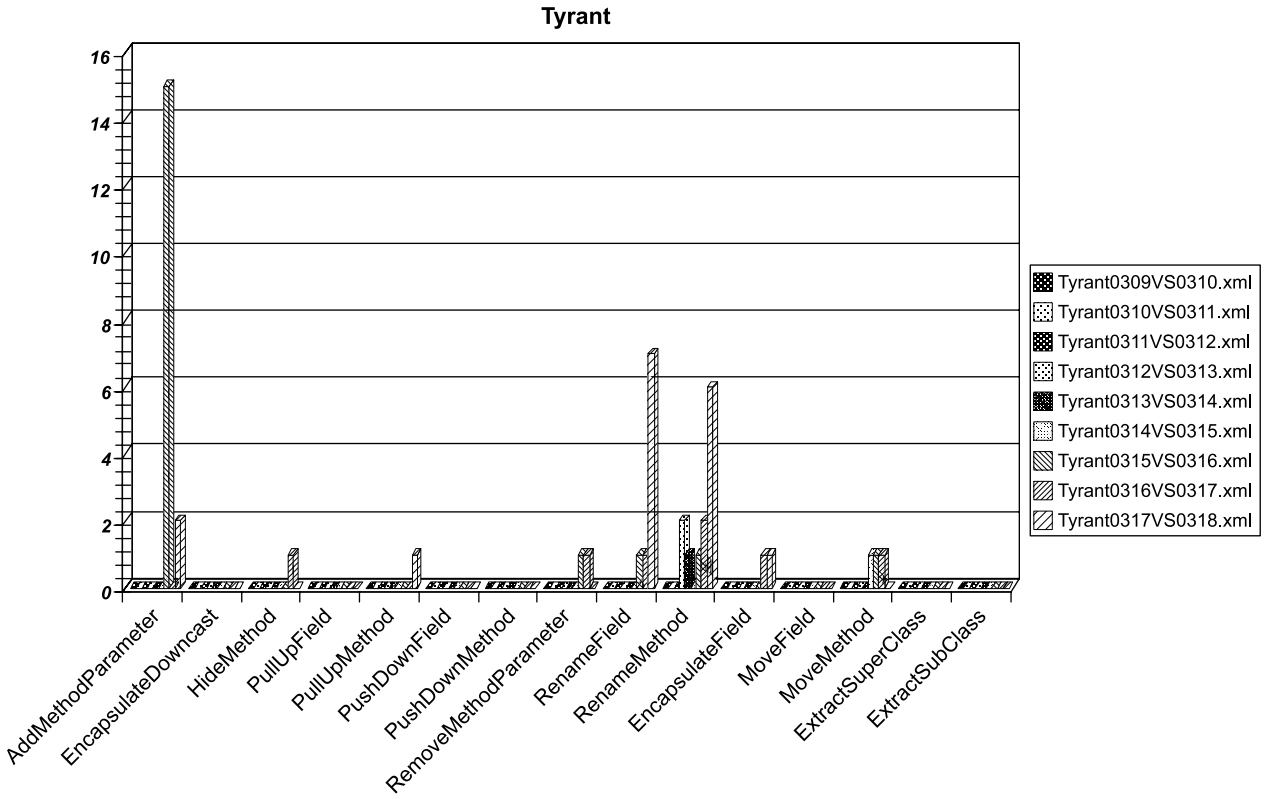

Figure 4. Refactorings for the tyrant system.

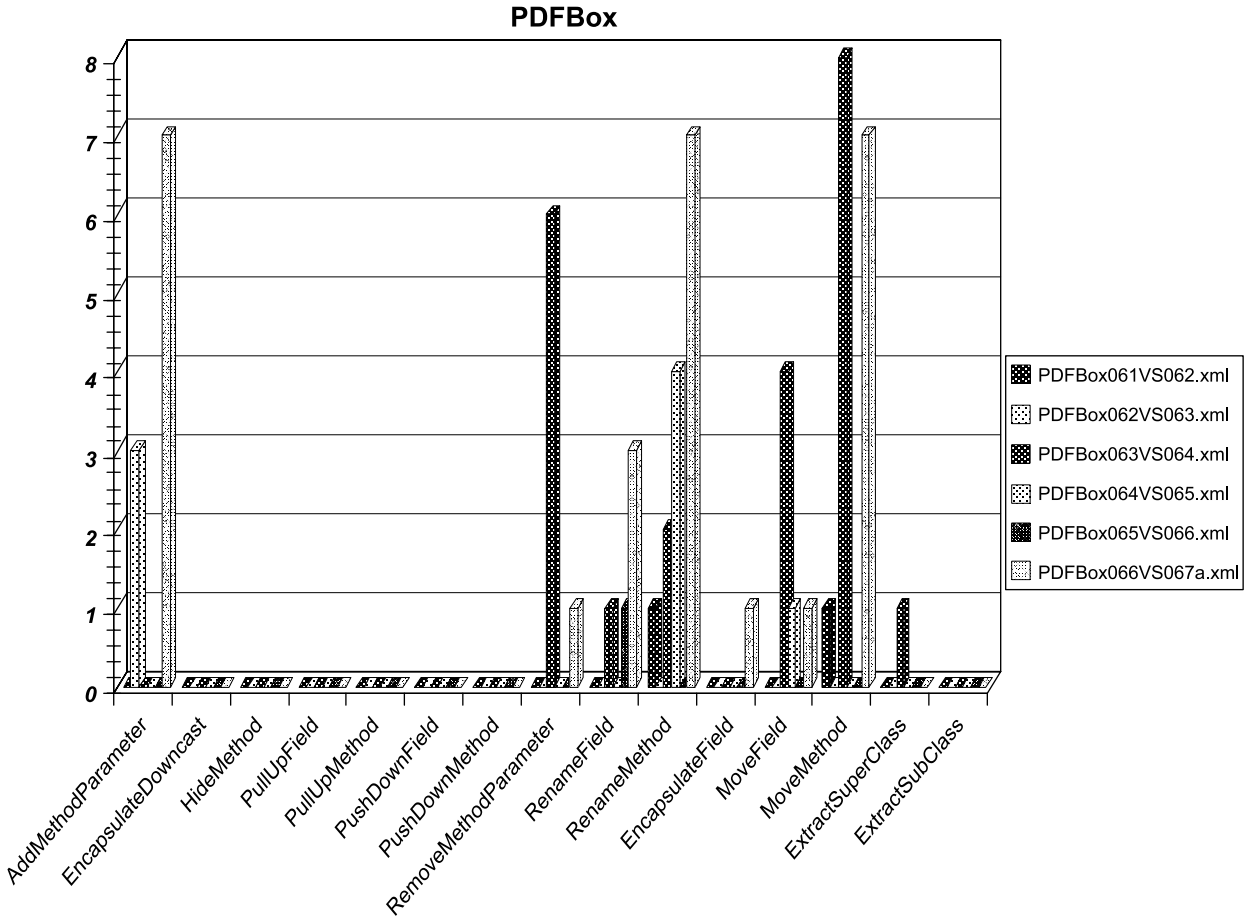

Figure 5. Refactorings for the PDFBox system.

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

Version	Megamek	Tyrant	Velocity	Antlr	HSQLDB	Jasper Reports	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	–	–	–	–	–

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 (Fowler 1999). 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 10 is the maximum number of versions (Tyrant); the contribution to table 3 of systems with fewer than 10 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 5–9, 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.

Refactoring type	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–9	9–10
Encapsulate downcast	0	0	0	0	0	0	0	0	0
Push down method	0	0	1	0	1	0	0	0	4
Extract subclass	0	2	3	0	1	0	0	0	0
Encapsulate field	0	5	4	0	0	1	0	1	1
Hide method	3	6	2	0	0	1	0	1	0
Pull up field	0	1	7	0	2	4	0	0	0
Extract superclass	0	2	10	0	8	1	0	0	2
Remove parameter	3	2	12	0	1	1	1	1	3
Push down field	0	16	3	0	7	0	0	0	0
Pull up method	0	9	17	0	24	5	0	0	10
Add parameter	13	16	41	1	1	9	15	0	3
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	137	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. This resulted in an analysis of the different dependencies between the 15 refactorings; in other words, when one refactoring is undertaken, does anther refactoring/refactorings necessarily need to happen? In the next section, we describe this dependency analysis from the data extracted by the tool.

4.3.1. A dependency analysis. As part of our ongoing refactoring research, we carried out a dependency analysis to establish the inter-relationships between the 72 refactorings. As a result of this analysis, 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

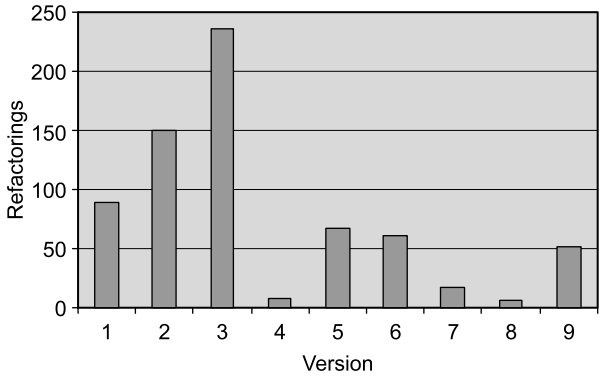

Figure 6. Total refactorings across versions.

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 restructuring 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 are not covered by the refactoring according to the same text.

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 15

refactorings in line with the mechanics specified by Fowler in Ferenc *et al.* (2004). However, we suggest that most of the simple refactorings were not as part of any larger refactoring.

While an analysis of the relationships between the different refactorings at a coarse level can reveal certain traits and relationships, we accept that a detailed treatment of each refactoring and its relationship with other refactorings may prove to be even more fruitful. Use of data mining techniques, for example the use of association rules and time series analyses in this sense would be of immense help and we leave such a treatment as an extension of this work. 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 (Ferenc *et al.* 2004). 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 72 refactorings, which we believed would provide a cross-section of the types of change typically made to software.

In our analysis we have not provided any analysis of the type of change undertaken—we have viewed each change as atomic, i.e. indivisible. We have also not addressed the average changes between releases as a mechanism for understanding the trends between versions. We view a treatment of these two features as 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 (O'Cinneide and Nixon 1998). 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. Finally, a more in-depth analysis of the relationship between refactorings through the use of data mining techniques such as association rules is planned.

Acknowledgements

The authors would like to thank the anonymous referees for their insightful and useful comments. The paper benefited significantly from these comments.

References

D. Advani, Y. Hassoun and S. Counsell, "Heurac: A heuristic-based tool for extracting refactoring data from open-source software versions", SCSIS-Birkbeck, University of London, Technical Report, BBKCS-05-03-01, 2005.

D. Arsenovski, Refactoring-elixir of youth for legacy VB code. Available at:www.codeproject.com/vb/net/Refactoring_elixir.asp.

K. Beck, "Extreme Programming Explained: Embracing Change", Glen View, IL: Addison Wesley, 2004.

L. Briand, C. Bunse and J. Daly, "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs", *IEEE Trans. Softw. Eng.*, 27(6), pp. 513–530, 2001.

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

S. Counsell, Y. Hassoun, R. Johnson, K. Mannock and E. Mendes, "Trends in Java code changes: the key identification of refactorings", in *ACM 2nd International Conference on the Principles and Practice of Programming in Java*, Kilkenny, Ireland, June 2003.

S. Demeyer, S. Ducasse and O. Nierstrasz, "Finding refactorings via change metrics", in *ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, Minneapolis, USA, 2000. pp. 166–177.

R. Ferenc, I. Siket and T. Gyimothy, "Extracting Facts from Open Source Software", in *Proceedings of 20th International Conference on Software Maintenance (ICSM 2004)*, Chicago, USA, 2004, pp. 60–69.

B. Foote and W. Opdyke, "Life cycle and refactoring patterns that support evolution and reuse", in *Pattern Languages of Programs*, James O. Coplien and Douglas C. Schmidt, Eds, Glen View, IL: Addison-Wesley, May, 1995.

M. Fowler, "Refactoring (Improving The Design of Existing Code)", Glen View, IL: Addison Wesley, 1999.

R. Harrison, S. Counsell and R. Nith, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems", *J. Sys. Softw.*, 52, pp. 173–179, 2000.

R. Johnson and B. Foote, "Designing reusable classes", *J. Obj. Orient. Program.*, 1(2), pp. 22–35, 1988. June/July.

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

T. Mens and T. Tourwe, "A survey of software refactoring", *IEEE Trans. Softw. Eng.*, 30(2), pp. 126–162, 2004.

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

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.

M. O'Cinneide and P. Nixon, "Composite refactorings for Java programs", in *Proceedings of the Workshop on Formal Techniques for Java Programs*, ECOOP Workshops, 1998.

W. Opdyke. "Refactoring object-oriented frameworks", PhD Thesis, University of Illinois. 1992.

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

D. Perry, "Laws and principles of evolution", Panel Paper, in *International Conference on Software Maintenance*, Montreal, Canada, 2002, pp. 70–71.

L. Tahvildari and K. Kontogiannis, "A metric based approach to enhance design quality through meta-pattern transformations", in *Proceedings of European Conference on Software Maintenance and Reengineering*, Benevento, Italy, 2003, pp. 183–192.
L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings", *Automated Softw. Eng*., 8 pp. 89–120, 2001.

Deepak Advani obtained a Master's Degree in Advanced Information Systems from Birkbeck, London in 2004. He is currently an independent researcher attached to the School of Computer Science and Information Systems at Birkbeck. Deepak has previously worked as a developer in the Software Industry. His research interests are in the Java programming language, refactoring and software tools.

Youssef Hassoun obtained his PhD from Birkbeck, London in 2005 investigating the reflection model in Java. He is currently a researcher in the School of Computer Science and Information Systems at Birkbeck. Previously Dr Hassoun has worked in the Software Industry as a developer and Project Manager. His research interests are Java and programming paradigms. Dr Hassoun also holds a PhD in Mathematical Physics from King's College, London.

Steve Counsell is a Lecturer in the School of Computing, Information Systems and Mathematics at Brunel University, which he joined in November 2004. Dr Counsell obtained his PhD in Software Engineering from Birkbeck, London in 2002 where he was a Lecturer. Between 1996 and 1998, Dr Counsell worked as a Research Fellow at Southampton University. Dr Counsell's research interests focus on metrics, refactoring and empirical studies.