A Meta-analysis Approach to Refactoring and XP

Steve Counsell, Robert M. Hierons,
Department of Information Systems and Computing,
Brunel University, Uxbridge, Middlesex. UB8 3PH.
{steve.counsell, rob.hierons}@brunel.ac.uk

George Loizou,
School of Computer Science, Birkbeck, University of London,
Malet Street, London, WC1E 7HX.
george@dcs.bbk.ac.uk

Abstract

The mechanics of seventy-two different Java refactorings are described fully in Fowler's text [13]. In the same text, Fowler describes seven categories of refactoring, into which each of the seventy-two refactorings can be placed. A current research problem in the refactoring and XP community is assessing the likely time and testing effort for each refactoring, since any single refactoring may use any number of other refactorings as part of its mechanics and, in turn, can be used by many other refactorings. In this paper, we draw on a dependency analysis carried out as part of our research in which we identify the 'Use' and 'Used By' relationships of refactorings in all seven categories. We offer reasons why refactorings in the 'Dealing with Generalisation' category seem to embrace two distinct refactoring sub-categories and how refactorings in the 'Moving Features between Objects' category also exhibit specific characteristics. In a wider sense, our meta-analysis provides a developer with concrete guidelines on which refactorings, due to their explicit dependencies, will prove problematic from an effort and testing perspective.

1. Introduction

As a software engineering discipline, refactoring has grown in prominence over the past few years [7, 8, 9, 10, 12, 15, 16, 17, 24]. Refactoring can be loosely defined as any change made to software in order to improve its structure without necessarily changing the semantics of the program. In principle, the consequent improvement in code comprehensibility makes the software easy to maintain and refactoring can provide both short-term and long-term benefits [21]. In fact, Fowler [14] suggests that the process of refactoring is the reversal of software 'decay' and any refactoring effort is worthwhile.

An open research problem in the refactoring and XP community [2] is establishing which of competing refactorings to undertake, based on the premise that a developer has only limited time for firstly, the refactoring activity and secondly, the subsequent testing required [4, 25, 26]. In this paper, we describe a dependency analysis in which we identify the relationships between refactorings in the seven categories originally specified by Fowler [13]. In other words, we investigated, for each refactoring X, which refactorings X 'Uses' as part of its mechanics, and equally, which refactorings X is 'Used By' as part of the mechanics of other refactorings. Our analysis examines the characteristics of these two relationships and, as a result, offers suggestions as to why certain categories of refactoring may be more problematic from a practical perspective.

2. Motivation and related work

A first motivation for our work is to highlight the features of individual refactorings and their inter-relationships [23, 27]. In particular, to highlight the difficulty of deciding on a specific refactoring when there may be many more hidden activities that only arise *during* a refactoring. Secondly, re-testing is a costly activity and to be undertaken properly requires an intimate knowledge of program refactoring mechanics. Those mechanics need to be investigated thoroughly before they can be used; if not, then those costs are likely to escalate. In this paper, we try to understand in more detail the implications of undertaking any refactoring activity.

The work described in this paper follows on from an earlier analysis by the same authors where an in-

1-4244-1031-2/07/$25.00©2007 IEEE

depth analysis of the refactoring trends (and of those fifteen refactorings) in Open-Source Systems (OSS) was documented [1]. Remarkably and surprisingly, inheritance and encapsulation-based refactorings were found to have been applied relatively infrequently from an empirical perspective, in keeping with past results relating to this OO concept [22]. The work in this paper also builds on other previous research by the authors where we investigated the link between refactoring and testing. In [5], we adapted a testing taxonomy proposed by Van Deursen & Moonen (VD&M) based on the post-refactoring repeatability of tests. The VD&M taxonomy proposed five categories of refactoring. In our assessment of the taxonomy, we urged the need for the inter-relatedness of refactorings to be considered when making refactoring decisions and we based that inter-relatedness on a refactoring dependency graph developed as part of the research. Given our taxonomy extension, we then assessed the potential for eliminating code smells [13] where minimum disruption to testing effort is the goal. Herein, we explore that inter-relatedness in greater detail.

In terms of broader related work, Najjar et al., have shown that an investigation of refactoring can deliver both quantitative and qualitative benefits [21] - the refactoring 'replacing constructors with factory methods' of Kerievsky [16] was used as a basis. 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. Developing heuristics for deciding on different refactorings, based on system change data, was earlier investigated by Demeyer et al. [8]. A study of the trends in changes, categorised according to refactorings was also undertaken in [7] and a full survey of relevant refactoring work can be found in [18].

3. The seven refactoring categories

Table 1 shows the seven categories of refactoring as defined by Fowler (with input from Beck) [13]. For simplicity, we have labelled these categories A-G. The 'Composing Methods' category (A) contains refactorings that package code up properly and place it where it fits most appropriately. For example, the 'Extract Method' refactoring takes one method whose purpose is not obvious and converts that method into two methods. The purpose of the 'Moving Features Between Objects' category (B) is to ensure, for example, that class coupling is minimized (e.g., by moving a method or field from one class to another). As its name suggests, the purpose of the refactorings in the 'Organising Data' category (C) is to ensure that data is declared, stored or manipulated in the most appropriate way. For example, the 'Encapsulate Field' refactoring changes the declaration of a field from public to private. The refactorings in the 'Simplifying Conditional Expressions' category (D) modify programmed conditions so that they are more understandable. The 'Decompose Conditional' refactoring for example, simplifies a conditional so that the 'condition' and 'else' components each have their own methods. The refactorings in the 'Making Method Calls Simpler' category (E) attempt to make the program interface easy to understand and straightforward in nature. The 'Rename Method' refactoring in this category, for example, renames a method so that its name expresses more clearly what the method does. Category F, 'Dealing with Generalisation' comprises refactorings related to manipulation of the inheritance hierarchy. Finally, the 'Big Refactorings' (Category G) are distinct from the other categories because as Fowler and Beck state, these refactorings take a long time to complete and *'... require a degree of agreement among the entire programming team that isn't needed with smaller refactorings'.*

Table 1. The seven categories and the refactorings they contain

Category	Refactorings
A: Composing Methods (9)	Extract Method, Inline Method, Inline Temp, Replace Temp with Query, Introduce Explaining Variable, Split Temporary Variable, Remove Assignments to Parameters, Replace Method with Method Object, Substitute Algorithm.
B: Moving Features Between Objects (8)	Move Method, Move Field, Extract Class, Inline Class, Hide Delegate, Remove Middle Man, Introduce Foreign Method, Introduce Local Extension.
C: Organising	Self Encapsulate Field, Replace Data Value with Object, Change Value to Reference, Change

Data (16)	Reference to Value, Replace Array with Object, Duplicate Observed Data, Change Unidirectional Association to Bidirectional, Change Bidirectional Association to Unidirectional, Replace Magic Number With Symbolic Constant, Encapsulate Field, Encapsulate Collection, Replace Record with Data Class, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Subclass with Fields.
D: Simplifying Conditional Expressions (8)	Decompose Conditional, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, Remove Control Flag, Replace nested Conditional with Guard Clauses, Replace Conditional with Polymorphism, Introduce Null Object, Introduce Assertion.
E: Making Method Calls Simpler (15)	Rename Method, Add Parameter, Remove Parameter, Separate Query from Modifier, Parameterize Method, Replace Parameter with Explicit Methods, Preserve Whole Object, Replace Parameter with Method, Introduce Parameter Object, Remove Setting Method, Hide Method, Replace Constructor with Factory Method, Encapsulate Downcast, Replace Error Code with Exception, Replace Exception with Test.
F: Dealing with Generalisation (12)	Pull Up Field, Pull Up Method, Pull Up Constructor Body, Push Down Method, Push Down Field, Extract Subclass, Extract Superclass, Extract Interface, Collapse Hierarchy, Form Template Method, Replace Inheritance with Delegation, Replace Delegation with Inheritance.
G: Big Refactorings (4)	Tease Apart Inheritance, Convert Procedural Design to Objects, Separate Domain from Presentation, Extract Hierarchy.

3.1. The dependency analysis

The purpose of our research is to investigate the implications in terms of required refactorings when deciding amongst *n* competing refactorings. To inform that analysis, a dependency list was developed providing, firstly, for each refactoring X, the list of the other *n* refactorings (where $n>=0$) that X 'Uses'. Secondly, the relationship between refactorings from the flip side; the 'Used By' relationship denotes, for each X, the *n* refactorings that use X as part of their mechanics. The dependency list was compiled by manually parsing all seventy-two refactorings in Fowler's text. For example, the 'Change Bidirectional Association to Unidirectional (CBAtoU)' refactoring 'Uses' two other refactorings as part of its mechanics. The purpose of the CBAtoU refactoring is to eliminate the two-way association between one class and another when one class no longer needs features of the other; a one-way association is created. The two refactorings used by CBAtoU are the 'Self Encapsulate Field' refactoring and the 'Substitute Algorithm' refactoring. Similarly, the 'Add Parameter' refactoring is 'Used By' only one refactoring, namely, the Introduce Parameter Object' refactoring. Equally, the 'Change Unidirectional Association to Bidirectional does not use any other refactorings as part of its mechanics.

For many of the seventy-two refactorings described in [13] the refactoring mechanics prescribe that a particular refactoring *must* use or *may* use refactoring X in order to be a successful refactoring itself. For example, the 'Introduce Parameter Object' refactoring, applicable when a group of parameters are lumped to form an object, requires the use of the Add Parameter refactoring in a 'must use' relationship in order that the new data clump can be formed. Equally, the 'Replace Data Value with Object' refactoring 'may use' the Change Value to Reference (CVtR) refactoring; we use this information in Section 4.

4. Data analysis

In the subsequent analysis, we first analyse the 'Uses' relationships amongst the seven categories (and then explore the 'Used By' relationship). We base our subsequent analysis on the premise that, other things remaining equal, the more refactorings that a refactoring 'Uses' as part of its mechanics, the greater the effort required by the developer to undertake the refactoring and the higher the testing burden as a result. Equally, refactorings with a high 'Used By' value (i.e., they are *used by* relatively high number of refactorings) suggests that those refactorings are 'core' refactorings, common to the mechanics (and the testing process) of many refactorings.

4.1 Summary data

Table 2 shows summary data for the dependencies amongst the seventy-two refactorings for both 'Uses' and 'Used By' relationships. It shows the maximum values in each case, together with the mean values for each. For example, in the 'Composing Methods'

category (A), the maximum number of 'Uses' by any one refactoring was 4, that pertaining to the 'Extract Method' refactoring; the maximum 'Used By' value in this category was 10, for the same refactoring. The mean value of 'Uses' dependencies was 0.78 and the mean of 'Used By' dependencies 2.44.

Table 2. Summary data for 'Uses' and 'Used By' refactorings

Category	Max 'Uses'	Max 'Used By'	Mean 'Uses'	Mean 'Used By'
A: Composing Methods	4	10	0.78	2.44
B: Moving Features Between Objects	3	12	1.13	2.75
C: Organising Data	3	7	1.38	1.06
D: Simplifying Conditional Expressions	4	3	1.25	0.75
E: Making Method Calls Simpler	3	9	0.73	1.4
F: Dealing with Generalisation	7	5	2.25	1.5
G: Big Refactorings	6	0	4.25	0

Two features of Table 2 are particularly noteworthy. Firstly, the relatively high 'Uses' mean value for Category F and G (2.25 and 4.25, respectively) and the relatively high 'Used By' mean values for Categories A and B (2.44 and 2.75, respectively). In the subsequent discussion, we elaborate at length on why some of these features are of relevance to the overall discussion.

4.2 'Uses' relationships

A key question which arises when a developer is deciding on a refactoring is the likely effort required for both the refactoring and the subsequent testing effort. To facilitate our analysis, we chose the set of fifteen refactorings with the highest 'Uses' values required as part of their mechanics. Figure 1 shows the frequency of 'Uses' relationships, ranging from 7 down to 3. We note that choosing the top ranked fifteen refactorings ensured that *all* refactorings with at least three 'Uses' dependencies were included in the analysis. (N.b., eleven refactorings had two 'Uses' dependencies, eighteen refactorings had a single 'Uses' dependency and the remaining twenty-eight refactorings had zero 'Uses' dependencies. In other words, none of those twenty-eight refactorings used any other refactorings.)

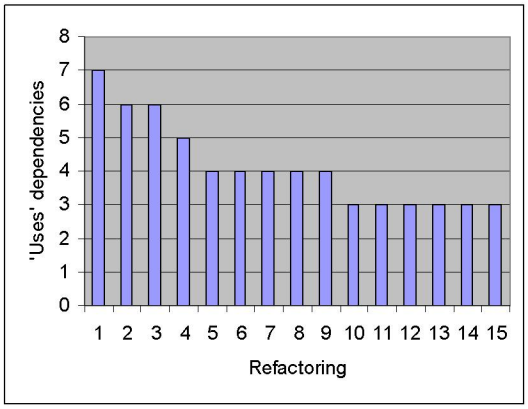

Figure 1. The fifteen refactorings with the highest number of 'use' dependencies'

The complete fifteen refactorings enumerated from Figure 1 in descending order of 'Uses' values are: 1. Extract Subclass (7), 2. Extract Superclass (6), 3. Extract Hierarchy (6), 4. Tease Apart Inheritance (5), 5. Collapse Hierarchy (4), 6. Extract Method (4) 7. Form Template Method (4), 8. Replace Conditional with Polymorphism (4), 9. Separate Domain From Presentation (4), 10. Change Value to Reference (3), 11. Encapsulate Collection (3), 12. Inline Class (3), 13. Introduce Parameter Object (3), 14. Replace Temp with Query (3), 15. Replace Type Code with Subclasses (3).

Table 3 shows the effect of placing each of these fifteen refactorings into their respective categories. It is interesting that a refactoring from each of the seven categories appears in Table 3. Most remarkable from the same table is the high proportion of refactorings taken from Categories C, F and G (3, 4 and 3, respectively); ten of the fifteen refactorings are taken from these three categories. More remarkable is the fact that, four of the top five refactorings from Figure 1 are drawn entirely from Category F (Dealing with Generalisation). Moreover, inspection of refactorings in Categories C and G reveal three more refactorings (i.e., Replace Type Code with Subclasses, Tease Apart Inheritance and Extract Hierarchy) to be *directly* related to manipulation of the inheritance hierarchy. This result suggests that any refactoring drawn from category F in Table 3 is likely to require the completion of a relatively large set of other refactorings through the 'Uses' relationship. Taking just categories F and G from Table 3, a question that arises is whether, for these seven inheritance-related refactorings, there is a strong intra-relationship? In other words, do those refactorings mainly 'Use' each other as part of their mechanics?

Table 3. The fifteen refactorings and their categories

Category	Refactorings
A: Composing Methods	Extract Method, Replace Temp with Query.
B: Moving Features between Objects	Inline Class.
C: Organising Data	Change Value to Reference, Encapsulate Collection, Replace Type Code with Subclasses.
D: Simplifying Conditional Expressions	Replace Conditional with Polymorphism.
E: Making Method Calls Simpler	Introduce Parameter Object.
F: Dealing with Generalisation	Extract Subclass, Extract Superclass, Collapse Hierarchy, Form Template Method.
G: Big Refactorings	Tease Apart Inheritance, Separate Domain from Presentation, Extract Hierarchy.

Table 4 shows the seven refactorings and, as part of their mechanics, the refactorings they in turn 'Use'. We have also included, for each refactoring, whether it is a 'may' or 'must' use relationship (as described in Section 3.1). To indicate a 'may' use relationship we have appended each relevant refactoring with an 'A' and to indicate a 'must' use relationship, appended each relevant refactoring with a 'U'. The bolded refactorings are those taken exclusively from 'Dealing with Generalisation' (Category F).

Table 4. The seven inheritance-related refactorings and their inter-relationships

Refactoring X	Refactorings that X 'Uses'
Extract Subclass (7)	Move Method (A), **Push Down Field (U)**, **Push Down Method (U)**, Rename Method (A), Replace Conditional with Polymorphism (A), Replace Constructor with Factory Method (A), Self Encapsulate Field (A).
Extract Superclass (6)	Form Template Method (A), **Pull Up Constructor Body (U)**, **Pull Up Field (U)**, **Pull Up Method (AU)**, Rename Method (A), Substitute Algorithm (A).
Collapse Hierarchy (4)	**Pull Up Field (U), Pull Up Method (U), Push Down Field (U), Push Down Method (U).**
Form Template Method (4)	Extract Method (U), Move Method (U), **Pull Up Method (U)**, Rename Method (U).

Replace Type Code with Subclasses (4)	**Push Down Field (U), Push Down Method (U)**, Self Encapsulate Field (U).
Tease Apart Inheritance (5)	Extract Class (U), Move Field (U), Move Method (U), **Pull Up Field (A), Pull Up Method (A).**
Extract Hierarchy (6)	Extract Class (A), Extract Method (A), Replace Conditional with Polymorphism (A), Replace Constructor with Factory Method (U), Replace Type Code with State/Strategy (A), Replace Type Code with Subclasses (A).

Table 4 exhibits a number of interesting properties. Firstly, there is a high dependence on the use of the Pull Up Method/Field and Push Down Method/Field refactorings. The motivation for the Pull Up Method refactoring is when two methods are doing the same thing and thus constitute duplication. Similarly for the Pull Up Field refactoring, (when two subclasses have the same field). In such a case, the field is simply moved to the superclass. The 'Collapse Hierarchy' refactoring uses these four refactorings exclusively and these four refactorings alone comprise thirteen of the thirty-six 'Uses' relationships. Only the 'Extract Hierarchy' refactoring fails to use any of these four refactorings. Secondly, there seems to be a high dependence on the use of 'Move Method' and 'Move Field' refactorings and, to a limited extent, the Extract Class/Method, Rename Method and Self Encapsulate Field refactorings. Fowler states [13] that the Move Method refactoring is the '*bread and butter of refactoring*'. Similarly, '*moving state and behavior between classes is the very essence of refactoring*'. Equally, 'Extract Method' is '*one of the most common refactorings I do*'. It comes as no surprise therefore to find these refactorings in Table 4.

In terms of the type of relationship (i.e., whether 'A' or 'U'), a clear pattern of 'U' relationships is evident for the Pull Up Method/Field and Push Down Field/Method refactorings. For the Extract Superclass refactoring, Pull Up Method engages in both an 'A' and a 'U' relationship. We interpret this feature to mean that the Extract Superclass must use the Pull Up Method refactoring as part of its mechanics, and in addition, there might be circumstances where there is a requirement for subsequent application of the same refactoring. Only for the 'Tease Apart Inheritance' refactoring is there any evidence of 'A' relationships for these four refactorings and there is a simple explanation to account for why the relationships are 'A' in type. The 'Tease Apart Inheritance' refactoring arises when the inheritance hierarchy is becoming 'spaghetti like' due to frequent addition of subclasses (and the creation of 'tangled inheritance'). In the words of Fowler: '*You have an inheritance hierarchy that is doing two jobs at once. Create two hierarchies and use delegation to invoke one from the other.*' The 'may' use relationship of Pull Up Field and Pull Up Method refactorings only arises *after* the main refactoring has been undertaken. According to Fowler '*Look at the new hierarchy for possible further refactorings such as Pull Up Method or Pull Up Field.*' The results suggest a strong intra-relationship between the refactorings in Category F.

4.3 'Used By' relationships

Figure 2 shows the converse set of values to that shown in Figure 1, ranging from 12 down to 3. In other words, it shows the highest ranked set of thirteen refactorings in terms of the 'Used By' dependency. We note that choosing the thirteen refactorings with the highest 'Used By' values ensured that *all* refactorings with at least three 'Used By' dependencies were included in Figure 2. (N.b., six refactorings had two 'Used By' dependencies, nineteen refactorings had just a single 'Used By' dependency and the remaining thirty-four refactorings had no 'Used By' dependencies; in other words, the latter were not used by any other refactoring).

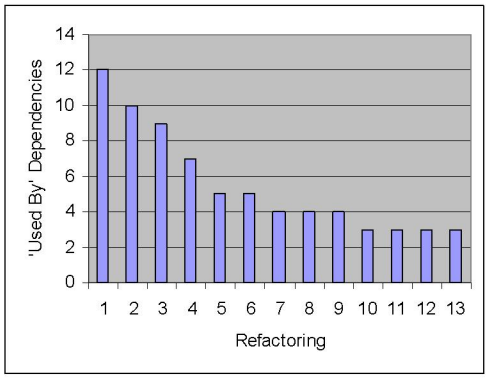

Figure 2. Thirteen refactorings with the highest 'Used By' dependencies

The thirteen refactorings shown in Figure 2 are 1. Move Method (12) 2. Extract Method (10), 3. Rename Method (9), 4. Self Encapsulate Field (7), 5. Pull Up Method (5), 6. Replace Constructor with Factory Method (5), 7. Move Field (4), 8. Pull Up Field (4), 9. Substitute Algorithm (4), 10. Push Down Field (3), 11. Push Down Method (3), 12. Extract Class (3), 13. Replace Conditional with Polymorphism (3).

Table 5 shows the effect of placing each of the fifteen refactorings from Figure 2 into their respective categories. The two categories that featured prominently in Table 4 (i.e., Dealing with Generalisation and Moving Features Between Objects) again feature heavily. Category G does not figure at all in this Table, in contrast to Table 3 and a simple explanation accounts for this trend: while the four refactorings in this category may 'Use' a number of refactorings, none of them are actually 'Used By' any other refactorings. This would make sense since they are 'big refactorings' which are more likely to 'Use' than be used by - the values in Table 2 reflect this bias. Inspection of the relevant data revealed the Tease Apart Inheritance refactoring to use five refactorings as part of its mechanics, Convert Procedural Design to Objects to use two refactorings, Separate Domain from Presentation to use four refactorings and Extract Hierarchy to use six.

In common with Table 3, Category F has the highest number associated 'Used By' refactorings (5) and surprisingly, these five refactorings are completely disjoint from the set of refactorings for the same category in Table 3. The evidence so far suggests that within Category F, there are two distinct types of refactoring. Firstly those that 'Use' many other refactorings as part of their mechanics and secondly, those that are 'Used by' many other refactorings. We can also observe an incestuous relationship between the 'Uses' set of refactorings and the 'Used By' refactorings within the same Category F (as evidenced in Table 4). This leads us to suggest the idea of a *client-server* relationship between the two disjoint sets of refactorings in this category - one set of refactorings is used frequently by the other set in a primarily servicing role. This was a surprising, yet interesting result to emerge from our analysis.

Table 5. The thirteen refactorings and their categories

Category	Refactorings
A: Composing Methods	Extract Method, Substitute Algorithm.
B: Moving Features Between Objects	Move Method, Move Field, Extract Class.
C: Organising Data	Self Encapsulate Field.
D: Simplifying Conditional Expressions	Replace Conditional with Polymorphism.
E: Making Method Calls Simpler	Rename Method, Replace Constructor with Factory Method.
F: Dealing with Generalisation	Pull Up Method, Pull Up Field, Push Down Field, Push Down Method.

The other category that features prominently in Table 5 and which also featured heavily in Table 4 is Category B refactorings. In common with the refactorings from Category F, the three refactorings are completely disjoint from those in the same category in Table 3, suggesting that these three refactorings are again engaged in some form of *client-server* relationship with other refactorings in the same category. However, there is a distinct difference between Category B and Category F refactorings. The refactorings from Category F in Table 5 only tend to be used by other Category F refactorings. On the other hand, the refactorings from Category B (Table 5) are used more widely by refactorings in other Categories (this can be inferred from their relatively high rankings in Figure 2). As further evidence of this feature, Table 6 lists the seven refactorings from Category B and F that appear in Table 5 and the refactorings they are 'Used By'. We annotate each set of refactoring with the category 'profile' reflecting the ordered list of categories from which those refactorings are taken. For example, the 'Move Field' refactoring is used by 4 refactorings, taken from categories B, B, G and G, respectively.

Table 6. The seven refactorings taken from Categories B and F and their profiles

Refactoring X	Set of refactorings that 'Uses' X
1. Move Method	Convert Procedural Design to Objects, Encapsulate Collection, Encapsulate Field, Extract Class, Extract Subclass, Form Template Method, Inline Class, Introduce Local Extension, Introduce Parameter Object, Replace Conditional with Polymorphism, Separate Domain From Presentation, Tease Apart Inheritance. **Profile**: G, C, C, B, F, F, B, B, E, D, G, G.
2. Move Field	Extract Class, Inline Class, Separate Domain From Presentation, Tease Apart Inheritance. Profile: B, B, G, G.
3. Extract Class	Extract Hierarchy, Hide Delegate, Tease Apart Inheritance. **Profile**: G, B, G.
4. Pull Up Method	Collapse Hierarchy, Extract Superclass, Form Template Method, Pull Up Constructor Body, Tease Apart Inheritance. **Profile**: F, F, F, F, G.
5. Pull Up Field	Collapse Hierarchy, Extract Superclass, Pull Up Method, Tease Apart Inheritance. **Profile**: F, F, F, G.
6. Push Down Field	Collapse Hierarchy, Extract Subclass, Replace Type Code with Subclasses. **Profile**: F, F, C.
7. Push Down Method	Collapse Hierarchy, Extract Subclass, Replace Type Code with Subclasses. **Profile**: F, F, C.

A number of points of interest can be drawn from Table 6. Firstly, eleven of the fifteen refactorings associated with the profile for refactorings 4, 5, 6 and 7 are drawn from category F. This is in contrast with just six out of a total of nineteen for Category B refactorings. There is a strong influence from Category G refactorings: five of the seven refactorings contain at least one Category G refactoring. Finally, it is interesting to note that the Push Down Field and Push Down Method refactorings are 'Used By' an identical set of three refactorings.

5. Conclusions and future work

In this paper, we have investigated the characteristics of the seventy-two refactorings and the categories into which each of those refactorings was placed. Results demonstrate some key traits in at least two of the categories described in [13]. To inform our analysis, we drew on a dependency analysis in which we identified the 'Use' and 'Used By' relationships amongst the seven categories of refactoring. Refactorings in the 'Dealing with Generalisation' category had two distinct refactoring types and refactorings in the 'Moving Features between Objects' showed specific characteristics. Our analysis provides a developer with information on which refactorings, due to their inherent dependencies, may prove to be more of a testing and maintenance burden than others. Consequently, our analysis can be used to inform difficult refactoring decisions that may ultimately be costly. In terms of future work, we intend to formally identify the relationships identified in this paper; we would also like to carry out more theoretical and empirical analyses to support or refute the arguments put forward in this paper in keeping with similar work in [3, 14, 20].

6. References

[1] D. Advani, Y. Hassoun and S. Counsell. Extracting Refactoring Trends from Open-source Software and a Possible Solution to the 'Related Refactoring' Conundrum. Proc. of ACM Symp. on Applied Computing, Dijon, France, April 2006.

[2] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.

[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] M. Bruntink and A. van Deursen. An empirical study into class testability. Journal of Systems and Software, 2006 (to appear).
[5] S. Counsell, R. M. Hierons, R. Najjar, G. Loizou and Y. Hassoun. The Effectiveness of Refactoring Based on a Compatibility Testing Taxonomy and a Dependency Graph. Proceedings of: Academic and Industrial Conference (TAIC PART), Windsor, UK, August 2006, pages 181-190. IEEE Computer Society Press.
[6] S. Counsell, P. Newson and E. Mendes, Architectural Level Hypothesis Testing through Reverse Engineering of Object-Oriented Software, Proc. of IEEE Int. Workshop on Program Comprehension, Limerick, Ireland, 2000.
[7] S. Counsell, Y. Hassoun, R. Johnson, K. Mannock and E. Mendes. Trends in Java code changes: the key identification of refactorings, ACM 2nd International Conference on the Principles and Practice of Programming in Java, Kilkenny, Ireland, June 2003.
[8] S. Demeyer, S. Ducasse and O. Nierstrasz, Finding refactorings via change metrics, ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Minneapolis, USA. pages 166-177, 2000.
[9] A. Van Deursen and L. Moonen. The Video Store Revisited - Thoughts on Refactoring and Testing. Proceedings of the third International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP 2002, Sardinia, Italy.
[10] A. van Deursen, L. Moonen, A. van den Bergh and G. Kok. Refactoring Test Code. In G. Succi, M. Marchesi, D. Wells, and L. Williams (eds.), Extreme Programming Perspectives. Addison Wesley, 2002, pages 141-152.
[11] T. Dinh-Trong and J. Bieman. Open Source Software Development: A Case Study of FreeBSD. Proceedings of 10th IEEE Intl. Symposium on Software Metrics, Chicago, USA, 2004, pp. 96-105.
[12] 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.
[13] M. Fowler. Refactoring (Improving the Design of Existing Code). Addison Wesley, 1999.
[14] 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.
[15] R. Johnson and B. Foote. Designing Reusable Classes, Journal of Object-Oriented Programming 1(2), pages 22-35. June/July 1988.
[16] J. Kerievsky, Refactoring to Patterns, Addison Wesley, 2004.
[17] T. Mens and A. van Deursen. Refactoring: Emerging Trends and Open Problems. Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE). University of Waterloo, 2003.
[18] T. Mens and T. Tourwe, A Survey of Software Refactoring, IEEE Transactions on Software Engineering 30(2): 126--139 (2004).
[19] A. Mockus, T. Fielding and D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 3, pages 309-346. 2002.
[20] S. Mouchawrab, L. C. Briand and Y. Labiche, A Measurement Framework for Object-Oriented Software Testability, Journal of Information and Software Technology, vol. 47, no. 15, pages 979-997, 2005.
[21] 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.
[22] R. Najjar, S. Counsell and G. Loizou. Encapsulation and the vagaries of a simple refactoring: an empirical study. Proceedings Int. Conference on Software Systems Engineering and its Applications, Paris, France, Dec. 2005.
[23] M. O'Cinneide and P. Nixon. Composite Refactorings for Java Programs. Proceedings of the Workshop on Formal Techniques for Java Programs. ECOOP Workshops 1998.
[24] W. Opdyke. Refactoring object-oriented frameworks, Ph.D. Thesis, Univ. of Illinois. 1992.
[25] M. Roper, Software Testing, McGraw-Hill, 1994.
[26] D. Saff, S. Artzi, J. Perkins and D. Ernst. Automatic test factoring for Java. Proceedings 21st Annual Int. Conference on Automated Software Engineering, Long Beach, USA, Nov. 9-11, 2005, pp. 114-123.
[27] T. Tourwe and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming, Proc. 7th European Conference on Software Maintenance and Re-Engineering, Benevento, Italy, 2003, pages 91-100.