The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph

S. Counsell and R. M. Hierons,
School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, Middlesex.
{steve.counsell, rob.hierons}@brunel.ac.uk

R. Najjar and G. Loizou,
School of Computer Science and Information Systems, Birkbeck, London.
rajaa@dcs.bbk.ac.uk

Y. Hassoun,
Dept. of Computing. Imperial College, London.
yhassoun@doc.ic.ac.uk

Abstract

In this paper, we describe and then appraise a testing taxonomy proposed by van Deursen and Moonen (VD&M) [9] based on the post-refactoring repeatability of tests. Four categories of refactoring are identified by VD&M ranging from semantic-preserving to incompatible, where, for the former, no new tests are required and for the latter, a completely new test set has to be developed. In our appraisal of the taxonomy, we heavily stress the need for the inter-dependence of the refactoring categories to be considered when making refactoring decisions and we base that need on a refactoring dependency graph developed as part of the research. We demonstrate that while incompatible refactorings may be harmful and time-consuming from a testing perspective, semantic-preserving refactorings can have equally unpleasant hidden ramifications despite their advantages. In fact, refactorings which fall into neither category have the most interesting properties. We support our results with empirical refactoring data drawn from seven Java Open-Source Systems (OSS) and from the same analysis form a tentative categorization of code smells.

1. Introduction

A key software engineering discipline to emerge over recent years is that of refactoring [2, 12, 15, 16, 18, 21, 22, 24]. Refactoring can be defined as a change made to software to improve its structure without necessarily changing the program's semantics. The 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. In the seminal text by Fowler et al., [13] it is suggested that refactoring is the reversal of software decay and, in that sense any refactoring effort is worthwhile. In the same text, seventy-two refactorings are proposed, all of which have specific mechanics and all of which incorporate re-testing along the way. For example, to 'rename a method' so that the purpose of the method is expressed more clearly, the method's name is changed and all references to the originally-named method are changed also; tests are carried out after every changed reference to ensure the refactoring has not been broken during this process.

A key assumption made about refactoring is that the external behaviour of the program does not change as a result of refactoring (only its structure changes); in this paper we take a different view and investigate, from a testing perspective, the features of all refactorings, many of which by their nature change the semantics of the program. We analyze a testing taxonomy proposed by van Deursen and Moonen (VD&M) [9] based on the post-refactoring repeatability of tests. The taxonomy of VD&M was informed by the difficulty of applying post-refactoring tests. In their words, refactoring a system should not: '*change its observable behaviour. Ideally, this is verified by ensuring that all the tests pass before and after a refactoring. In practice, it turns out that such verification is not always possible: some refactorings restructure the code in such a way that tests can only pass after the refactoring if they are modified.*'

In our analysis, we postulate that the inter-relatedness (and hence the dependencies) between refactorings needs to be of paramount consideration when making refactoring decisions - we base that inter-relatedness on a refactoring dependency graph developed as part of the research. Given our taxonomy analysis, we then assess a set of empirical refactoring data extracted from seven Java OSS and, based on the same analysis, the potential for eliminating code smells [13] where minimum disruption to testing effort is the goal. Results indicate that adopting a framework such as that proposed by VD&M firstly, provides a useful starting point for developers and project managers alike for making decisions on which refactorings are preferable from a testing perspective and secondly, can inform our understanding about empirical data and code smells.

2. Motivation and related work

The motivation for this research stems from two sources. Firstly, most definitions of refactoring are expressed in terms of semantic-preserving operations (i.e., the external behaviour of a system stays the same – its internal structure is what changes). Surprisingly, very little theoretical or empirical work has focused on the practical limitations imposed on developers by refactorings that explicitly change the program interface. In this paper, we are able to assess the testing implications of such refactorings through VD&M's taxonomy. Secondly, the link between testing, itself the subject of much work [20, 25], and refactoring permeates Fowler's text; again, surprisingly little work has investigated the *formal* link between the two. In VD&M's paper [9], and upon which the work in this paper is based, the taxonomy is described and the concept of a refactoring 'session' which uses knowledge about the link between testing and refactoring to inform changes in both is introduced. The authors coin the term 'test-first refactoring' to mean refactoring 'which uses the existing test cases as the starting point for finding suitable code level refactorings'; we describe the finer details of the taxonomy in more detail later in this paper. A number of other works have also investigated the related topics of class testability and refactoring of the test code itself [5, 10, 26].

In terms of other related work, Najjar et al., has shown that refactoring can deliver both quantitative and qualitative benefits [21] - the refactoring 'replacing constructors with factory methods' of Kerievsky [16] 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. Herein, we relate the testing taxonomy to fifteen specific refactorings. An in-depth analysis of the refactoring trends (and of those fifteen refactorings) in OSS was originally provided in [3]. Results showed the most common refactorings of the fifteen we term the 'Gang of Six', to be generally those with a high in-degree and low out-degree when mapped on a dependency graph; the same refactorings also featured strongly in the remedying of bad code smells. Remarkably and surprisingly, inheritance and encapsulation-based refactorings were found to have been applied relatively infrequently. The paper thus identified 'core' refactorings central to many of the changes made by developers of open-source systems. A 'peak' and 'trough' effect in the pattern of refactorings was observed across all but one of the systems studied, suggesting that refactoring is done in effort 'bursts'. 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. van Deursen & Moonen's (VD&M's) taxonomy

According to van Deursen and Moonen [9], henceforward referred to as 'VD&M', extreme programmers improve the design of their programs through constant refactoring - in Extreme Programming (XP) [3], tests are fully automated and documented. The analysis in this paper is of a refactoring taxonomy proposed by VD&M based on the impact that a refactoring has on the ability to use the same set of tests 'post-refactoring'. In other words, to what extent can we use the same test 'set' after refactoring? Does the test set need to be extended (or modified); if the latter, to what extent? The taxonomy developed by VD&M is motivated as follows: '*One of the dangers of refactoring is that a programmer unintentionally changes the systems' behavior. Ideally, it can be verified that this did not happen by checking that all the tests pass after refactoring. In practice however, there are refactorings that will invalidate tests (e.g., when a method is moved to another class and the test still expects it in the original class).*'

VD&M distinguish between two types of refactoring; firstly, refactorings that *do not* change an interface of the classes of the system and secondly, refactorings that *do* change an interface of the classes of a system. The first type of refactoring does not affect the set of tests required as a result of the refactoring (since the refactoring preserves tested behaviour). The second type of refactoring is one that can have consequences for the test set, since those tests still expect the old interface rather than the new one). They thus define two categories of refactoring:

1. *Incompatible*: The refactoring destroys the original interface. All tests which rely on the old interface must be adjusted in some way to accommodate the new interface.
2. *Backwards Compatible*: The refactoring extends the original interface. The tests keep running via the original interface and will pass if the refactoring preserves tested behavior. Depending on the type of refactoring, more tests may need to be added to cover the extensions.

VD&M describe four separate categories into which each of the seventy-two refactorings of Fowler fall. Originally, five categories were described in their paper – including a *composite* refactoring category. These 'Type A' refactorings were dropped from their analysis on the basis that the 'big four' refactorings comprising this category were 'performed as a series of smaller refactorings' and could not be analyzed in the same way as the other four categories. We are thus left with sixty-eight different refactorings for our analysis. The four remaining categories (and consequently, the four we adopt in this paper) are:

1. *Compatible:* Refactorings that do not change the original interface. Henceforward, we refer to these as Type B refactorings.
2. *Backwards Compatible:* Refactorings that change the original interface and are inherently backwards compatible since they extend the interface. Henceforward, we refer to these as Type C refactorings.
3. *Make Backwards Compatible:* Refactorings that change the original interface and can be made backwards compatible by adapting the old interface. For example, the 'Move Method' refactoring that moves a method from one class to another can be made backwards compatible through the addition of a 'wrapper' method to retain the old interface. (A wrapper is an object capable of transforming the external view that an interface shows in some way.) Henceforward, we refer to these as Type D refactorings.
4. *Incompatible:* Refactorings that change the original interface and are not backwards compatible because they may, for example, change the types of classes that are involved making it difficult to wrap the changes. Henceforward, we refer to these as Type E refactorings.

Type B and C refactorings from the sixty-eight refactorings according to VD&M are listed in Table 1. In theory, a developer should always prefer type B refactorings in preference to any other refactoring Type since no change to the test suite is required after those refactorings have been completed. Alternatively, type C refactorings are still feasible and, in theory, more desirable than Type D or E refactorings, but they may still require extensions to the test set after completion.

Table 1. Type B and Type C refactorings

Type B	Change Bi-directional Association to Unidirectional, Replace Magic Number with Symbolic Constant, Replace Nested Conditional with Guard Clauses, Consolidate Duplicate Conditional Fragments, Replace Conditional with Polymorphism, Replace Delegation with Inheritance, Replace Inheritance with Delegation, Replace Method with Method Object, Remove Assignments to Parameters, Replace Data Value with Object, Introduce Explaining Variable, Replace Exception with Test, Change Reference to Value, Split Temporary Variable, Decompose Conditional, Introduce Null Object, Preserve Whole Object, Remove Control Flag, Substitute Algorithm, Introduce Assertion, Extract Class, Inline Temp.
Type C	Consolidate Conditional Expression, Replace Delegation with Inheritance, Replace Inheritance with Delegation, Replace Record with Data Class, Introduce Foreign Method, Pull Up Constructor Body, Replace Temp with Query, Duplicate Observed Data, Self Encapsulate Field, Form Template Method, Extract Superclass, Extract Interface, Push Down Method, Push Down Field, Extract Method, Pull Up Method, Pull up Field.

We note that both 'Replace Delegation with Inheritance' and 'Replace Inheritance with Delegation' appear in both Type B and Type C categories, since they can be in either category depending on the case being considered [9]. The type D and E refactorings again according to VD&M are listed in Table 2. Developers would want to avoid Type E refactorings at all costs since they destroy the original interface and thus require large-scale changes to the test set. Type D refactorings can be made backwards compatible and would thus be preferable to refactorings of type E, yet may still require a modification to the code by introduction of the wrapper as previously described.

Table 2. Type D and Type E refactorings

Type D	Change Unidirectional Association to Bi-directional, Replace Parameter with Explicit Methods, Replace Parameter with Method, Separate Query from Modifier, Introduce Parameter Object, Parameterize Method, Remove Middle Man, Remove Parameter, Rename Method, Add Parameter, Move Method.
Type E	Replace Constructor with Factory Method, Replace Type Code with State/Strategy, Replace Type Code with Subclasses, Replace Error Code with Exception, Replace Subclass with Fields, Replace Type Code with Class, Change Value to Reference, Introduce Local Extension, Replace Array with Object, Encapsulate Collection, Remove Setting Method, Encapsulate Downcast, Collapse Hierarchy, Encapsulate Field, Extract Subclass, Hide Delegate, Inline Method, Inline Class, Hide Method, Move Field.

One interesting observation from Table 1 is the high number of inheritance-related refactorings in the Type C category and the emphasis on parameter manipulation in the Type D category. Inheritance relationships are relatively easy to preserve through refactoring since operations of pulling up and pushing down class features, for example, can be maintained through inheriting any behaviour passed up the hierarchy. Equally, behaviour pulled down is only normally pulled down because it was being used by the subclass anyway. Manipulation, addition and removal of method parameters (common to Type D refactorings) can be relatively easily masked by the wrappers (hence their inclusion as refactorings that can be made backwards compatible).

Also noteworthy is the high proportion of encapsulation-based and information hiding-oriented refactorings in Type E. In one sense, this destructive nature of the refactoring highlights the expressive power of encapsulation/information hiding in completely changing the program's semantics for the better. While this may not be beneficial from a tester's viewpoint because old tests will no longer apply, added encapsulation/information hiding does tend to improve program OO robustness through its enforced semantics.

4. A Dependency graph

As part of our refactoring analysis and to inform our understanding of the VD&M taxonomy, we developed a 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 relationship between those refactorings given by the mechanics of each refactoring in [1]. 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. The graph took three months to develop and was extracted using Fowler's text. A distinction was made on the graph between 'must use' relationships and 'may use' relationships to recognize the relative influence of each; these relationship types we now explain.

4.1 'Must Use' relationships

One issue which arises as a result of the VD&M taxonomy is the dependence of one refactoring on one or more other refactorings, an open problem acknowledged in [9]. For many of the seventy-two refactorings described in [13] the refactoring mechanics prescribe that a particular refactoring *must* use refactoring *x* in order to be a successful refactoring [17]. For example, from the dependency graph, 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 for the new data clump so formed.

This 'must use' relationship has significant implications for the taxonomy proposed by VD&M. For example, while it would not be problematic for a refactoring; let's say, *n* of Type E to use any number of refactorings of Type B, C or D 'higher up' the taxonomy, any refactoring using a refactoring in the opposite direction i.e., 'lower down' in the taxonomy may cause a problem. (We would view Types C, D and E as 'lower down' than Type B and similarly for the relationships between C, D and E Type refactorings.) If a type B refactoring 'must use' a Type C, D or E refactoring then that immediately invalidates the semantics of Type B refactorings since the assumption for type B refactorings is that they do not change the interface. For a Type B refactoring, inclusion of a refactoring from either of types C, D E has that effect. Moreover, the use of a refactoring of Type C, D or E may also cause a chain effect requiring still further refactorings to be needed of Types 'lower down' in the taxonomy. For example, the Extract Class refactoring (Type B) must use the 'Move Field' refactoring of Type E and also the 'Move Method' – a Type D refactoring as part of its mechanics. In Fowler's words for the Extract Class refactoring: '*Use Move Field on each field you wish to move*' and '*Use Move Method to move methods over from old to new*'.

The 'Form Template Method' refactoring (Type C) also uses the Rename Method (RM) refactoring - a Type D refactoring. At the other end of the scale, the 'Introduce Null Object' – a Type B refactoring uses zero other refactorings as part of its mechanics and, as such, can be used with impunity post-refactoring using the original test set.

4.2 'May Use' relationships

Just as 'must use' relationships specify refactorings that must be undertaken to facilitate another refactoring, other refactorings 'may' require the use refactorings drawn from other Types. Despite this relationship being conditional, the issue described in Section 4.1 may still emerge if the conditions hold during refactoring. For example, the 'Replace Data Value with Object' refactoring (Type A) may use the 'Change Value to Reference' (CVtR) refactoring (Type E). According to Fowler, the motivation for using the CVtR refactoring is when '*you have a data item that needs additional data or behaviour*'. The data item is turned into an object as a result. The example for this refactoring given is that of a telephone number which needs extra behaviour for formatting, extracting the area code etc. As part of the mechanics of the refactoring, the developer '*may need to use CVtR on the new object*'.

4.3 Relationship chains

For both the 'must use' and 'may use' relationships, a far more sinister relationship exists bonding the two. Often, a 'chain' of refactorings will occur from a single refactoring implying that, for example, a Type B refactoring may use a Type E refactoring which itself must use a further Type E refactoring. A chain of refactorings then needs to be followed. The implications for testing thus go beyond the requirements of the first 'link' in the chain. For example, continuing the example from Section 4.2, the CVtR refactoring *must* use the 'Replace Constructor with Factory Method' refactoring which *may* then use the RM refactoring (Type D). The RM may itself use the 'Add Parameter' refactoring (Type D) and may also use the 'Remove Parameter' refactoring (Type D).

For each of the sixty-eight refactorings in Tables 1 and 2, we can, using the dependency graph, easily identify the Type B and Type C refactorings that break the test semantics through the use of Type D or E refactorings. We first investigate the research questions: *Which Type B refactorings use no other refactorings as part of their mechanics or only use other Type B refactorings as part of their mechanics? Which Type C refactorings use no other refactorings, other than Type B or Type C refactorings?*

Any refactoring in the first category can be undertaken safely in the knowledge that the original test set does not need to be modified. Table 3 shows the refactorings from Table 1 for Type B refactorings that *do not* use any 'lower down' Types of refactoring (i.e., of Types C, D or E) whether of 'may' or 'must' use relationship types. Equally, for Type C refactorings, we give refactorings which do not use any other lower Type of refactoring (i.e., Types D or E); they may possibly use 'higher' Types (i.e., those of Type B). We have also eliminated any refactoring which indirectly uses a non-Type B refactoring in the former case, or any refactoring that indirectly uses a non-Type C refactoring in the latter case.

Table 3. Type B refactorings (preserving tests) and Type C refactorings (extending tests)

Type B	Consolidate Duplicate Conditional Fragments, Replace Inheritance with Delegation, Remove Assignments to Parameters, Introduce Explaining Variable, Replace Exception with Test, Introduce Null Object, Substitute Algorithm, Inline Temp.
Type C	Replace Inheritance with Delegation, Self Encapsulate Field, Extract Interface, Push Down Method, Push Down Field, Extract Method, Pull Up Method, Pull Up Field.

From an initial set of twenty-two refactorings in Type B in Table 1, only eight can be safely used on the original test set; for Type C, only eight of an initial seventeen remain. Interestingly, in the new Type B list, there is a high number of refactorings relating to changes in the way the code is explicitly written rather than in the manipulation of objects. For example, the 'Substitute Algorithm' refactoring is applicable when '*you want to replace an algorithm with one that is clearer'*. Equally, the 'Consolidate Duplicate Conditional Fragments' refactoring is applicable when '*The same fragment of code is in all branches of a conditional expression*'. Six of the eight remaining refactorings fall into this category (the two other refactorings are 'Replace Inheritance with Delegation' and 'Introduce Null Object'). As interesting to note from Table 3 is the high number of inheritance-based refactorings remaining of Type C (five of the eight relate directly to operations on the inheritance hierarchy).

To complete the picture, Table 4 shows the number of D type refactorings which only use their own type of refactoring or lower down (i.e., those of type B or C). Remarkably, inspection of the dependency graph revealed that only one type D refactoring used any Type E refactoring (i.e., Replace Parameter with Method). A simple explanation accounts for this result. According to the dependency diagram, Type D refactorings are predominantly refactoring 'sinks'. In other words, they are *used by* a large number of other refactorings (i.e., they have a high in-degree) but do not tend to use other refactorings themselves (i.e., they have an out-degree of mostly zero and occasionally one).

Table 4. Type D refactorings that use only Type D, C or B refactorings

Type D	Change Unidirectional Association to Bi-directional, Replace Parameter with Explicit Methods, Separate Query from Modifier, Introduce Parameter Object, Parameterise Method, Remove Middle Man, Remove Parameter, Rename Method, Add Parameter, Move Method.

We could thus view Type D refactorings as preferable to any Type B or Type C refactoring which use a Type E refactoring on the single basis that Type D refactorings can be made backwards compatible and Type E refactorings are incompatible. Moreover, as well as a tendency for Type D refactorings to be sinks, the same set of refactorings also contain a high number of 'isolated' refactorings (i.e., those that use no other refactorings whatsoever). The same is not true of type B, C or E Type refactorings. We omit a similar analysis of the Type E refactorings on the basis that whatever other refactorings they use will not assist the tester in any sense (since incompatibility of the refactoring will always be maintained).

4.4 Implications of the results

A number of implications arise from these results. Firstly, a Type B refactoring should only be chosen if it is taken from the top half of Table 3 thereby preserving the original interface. Equally, a Type C refactoring should only be chosen if it is taken from the bottom half of Table 3 for the same reason. Any Type D refactoring or combination thereof can be chosen at will with the knowledge that it will not generally require any other refactorings to be applied as a result (the one exception being the IPO refactoring). Finally, no Type E refactorings should be undertaken, for the good reason that VD&M suggest (i.e., that they destroy the original interface).

5. Two follow-up investigations

As a result of the analysis in Section 4, we now provide details of two subsequent investigations; the first looks at empirical data of extracted refactorings from seven Java OSS. A research question that arises from our analysis is whether, if what we suggest is true, empirical data on refactoring shows that developers will avoid Type E refactorings in favour of Type D refactorings. Secondly, we look at the impact that our analysis has had on the elimination of bad code smells.

5.1 Empirical data

Figure 1 shows in ascending order, the average frequency over all versions for fifteen refactorings extracted from seven Java OSS applications [11,19]. The refactorings were extracted using a set of heuristics embedded in the tool for each refactoring. The tool itself and an in-depth analysis of the fifteen refactorings extracted are described in more detail in [1] and for space considerations herein we direct the interested reader to this reference. The fifteen refactorings in the order from left to right are 1) Encapsulate Downcast, 2) Push Down Method, 3) Extract Subclass, 4) Encapsulate Field, 5) Hide Method, 6) Pull Up Field, 7) Extract Superclass 8) Remove Parameter, 9) Push Down Field, 10) Pull Up Method, 11) Move Method, 12) Add Parameter, 13) Move Field, 14) Rename Method and 15) Rename Field. Refactoring 1 (with zero occurrences was 'Encapsulate Downcast') and the most frequent refactoring extracted by the tool - 'Rename Field'.

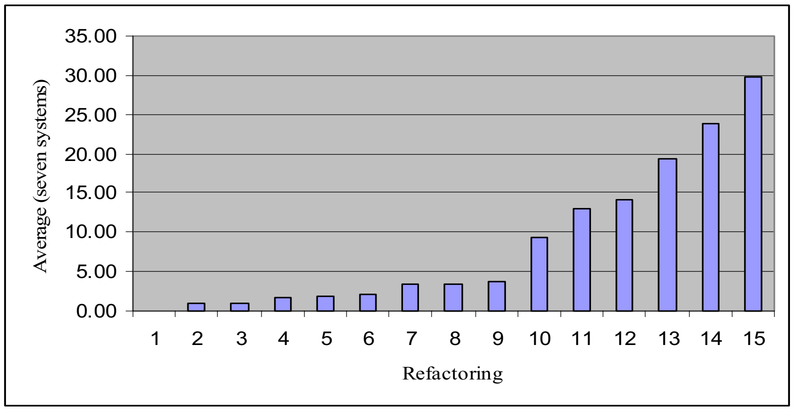

Figure 1. Frequency of fifteen refactorings extracted from seven Java OSS

Table 5 identifies each of the fifteen refactorings together with the Type they fall into according to the VD&M taxonomy, the position of the refactoring according to its frequency and the number of occurrences extracted for that refactoring. For example, the Pull Up Method refactoring belongs to Type C and was refactoring number 10 from Figure 1 with 65 occurrences. Interestingly, the most popular refactorings are those from Type D, where the most popular, second, fourth and fifth most popular refactorings are found. The least common refactorings appear to be those from Type E, which contained the least popular refactoring and the third, fourth and fifth least popular refactorings. Type C refactorings fell somewhere in the middle and had the second least, sixth, seventh, ninth and tenth least popular refactorings.

We note that the tool did not extract any refactorings from Type B for one simple reason. The complexity of the heuristics for a Type B refactoring make it impossible for such a refactoring to be identified through syntax alone. To guarantee automatic identification of a refactoring such as 'Substitute Algorithm', the syntax of the code would have to be parsed for any differences and then the semantics of the code examined to ensure that program meaning had not changed. On the other hand, renaming and hiding of fields and/or methods are relatively simple refactorings, do not require major program changes and can be identified syntactically by a tool relatively easily. Other refactorings, such as extracting sub- and/or super classes are more complex, require structural changes involving the class hierarchy and while more involved in terms of how they were implemented by the tool, can be encoded relatively easily. The poor showing of inheritance related refactorings (numbers 2, 3, 6, 7, 9 and 10 on Figure 1) was interesting; one explanation for this may be that modifying the inheritance structure is too 'revolutionary' in impact and is left alone in favour of more simple refactorings that do not perturb the high-level design [4, 6, 14].

We can draw a number of conclusions from the evidence in Table 5. Firstly, Type D refactorings appear to be the most popular and Type E the least popular refactoring. Interestingly, the only Type C refactoring that figures in the top five refactorings is the Pull Up Method refactoring. The obvious question that arises from Table 5 is whether the most popular refactorings taken from Type D had that property because they were 'used' a large number of times by other Type E refactorings. For example, the 'Extract Subclass' refactoring may Rename multiple Methods so forming a '1:many' relationship. However, no evidence whatsoever could be found on inspection of the dependency diagram to link any Type E refactorings to the Type D refactorings of Table 5 even through an indirect relationship. Interestingly, the main use of Type D refactorings came from those of Type C. From our analysis, we could conclude that empirically, Type E refactorings are avoided perhaps for the reasons that VD&M state in their paper.

Table 5. The fifteen refactorings and the category Types they fall into

Type	Applicable Refactorings
B	None
C	Pull Up Method (10-65), Push Down Field (9-26), Extract Superclass (7-23), Pull Up Field (6-14), Push Down Method (2-6),
D	Move Method (11-88), Add Parameter (12-99), Rename Method (14-167), Rename Field (15-200), Remove Parameter (8-24),
E	Move Field (13-135), Hide Method (5-13), Encapsulate Field (4-12), Extract Subclass (3-6), Encapsulate Downcast (1-0)

A further research question that arises is whether elimination of code 'smells' is hampered or assisted by the use of certain Types of refactoring (in terms of subsequent required testing). We address this question in the next section.

5. 2 The role of code smells

According to Fowler, bad smells in code are structures in the code that '*sometimes scream for*' the possibility of refactoring. A bad smell in code should thus be the key impetus for undertaking refactoring effort. Ideally, we want to eliminate code smells that involve the least re-testing effort. Consequently, if we can identify smells that require at least one Type E refactoring, then we could begin by ignoring that smell in favour of an alternative which requires combinations of only Type B and C refactorings.

Our analysis revealed nine bad smells in code from the twenty-two in total specified by Fowler to *not* contain a refactoring of Type E – and thirteen contained at least one Type E refactoring. Table 6 lists the two smells of those nine whose remedies are taken exclusively from Table 3 or 4. For example, the first bad smell, 'Alternative classes with different interfaces' refers to a smell where many methods are doing the same thing but with different signatures – such methods can be optimized. This code smell is eliminated through the use of the Rename Method and Move Method refactorings, both of which are taken from the Type D refactorings (Table 4).

Table 6. Bad Smells in code and the refactorings that remedy those refactorings

Bad Smell (BS)	Refactorings Remedies	Type Profile
Alternative Classes with Different Interfaces	Rename Method, Move Method	**D, D**
Refused Bequest	Replace Inheritance with Delegation	**B/C** (remedy appears in both categories)

If we adopt the rule that we should look to eliminate code smells on a Type scale, then an undesirable smell to remedy will be one whose profile uses at least one Type E refactoring. Next, we should avoid any smell whose profile includes a Type B or C refactoring not found in Table 3. The two bad smells shown in Table 6 therefore represent the only two smells for which we can guarantee there are no chains (because they are all of Type D refactorings), or alternatively whose refactorings are taken exclusively from Table 3, namely the Refused Bequest (RB) smell. The RB bad smell refers to a situation where a subclass or subclasses do not need the class features of the super class and as such, a new sibling class needs to be created to accommodate the 'refused' behaviour. Table 7 illustrates some of the least remediable smells of the thirteen containing at least one Type E refactoring.

Table 7. A sample of the least desirable smells

Bad Smell (BS)	**Refactorings Remedies**	**Type Profile**
Primitive Obsession	Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy	**B, B, D, E, E, E, E**
Data Class	Move Method, Encapsulate Field, Encapsulate Collection	**D, E, E**

The Primitive Obsession smell arises with an obsessive over-use of primitive data types in classes; the Data Class smell arises when a class has just getting and setting methods and nothing else. Such behaviour should try to be accommodated elsewhere. Sadly, inspection of all twenty-two bad smells revealed Type E refactorings to occur the most frequently and to appear in the most number of smells. We conclude that while eradication of code smells is a useful technique to adopt as part of an XP strategy, care should be taken to avoid those smells that will cause significant amounts of effort and re-testing.

6. Conclusions and future work

In this paper, we have analysed a testing taxonomy originally proposed by van Deursen and Moonen, in which they describe a distinct set of categories for post-refactoring assessment. The taxonomy provided a valuable framework from which further analysis could be made and in turn allowed us to investigate the properties of a dependency diagram showing the relationships between the seventy-two refactorings originally proposed by Fowler. We supported our analysis with some empirical data from a previous study of Java OSS and demonstrated that while semantic preserving refactorings may be ideal for preserving test sets, they are not necessarily always the *right* refactorings to choose. We postulated that the choice of refactorings should be based predominantly on the extent of inter-relatedness of refactorings and that this choice also extends to the elimination of code smells. Developers should not undertake refactorings based on their superficial characteristics but look more carefully into the mechanics of the refactorings they intend to undertake.

In terms of the implications of our analysis, we see our results as being of use to developers when deciding amongst competing and often composite refactorings [23] in the context of limited available maintenance and testing time and also amongst the different smells emerging from code as it evolves. In terms of future work, we will look into the possibility of a developer opinion based study of refactorings and code smells; in other words, which refactorings do developers prefer doing (if any) and also what code smells do they prefer to eradicate. We would also like to investigate the link between testing, refactoring and the incidence of faults found in OSS.

7. 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. To appear in the Proceedings of ACM Symposium on Applied Computing, Dijon, France, April 2006.
[2] D. Arsenovski. Refactoring– elixir of youth for legacy VB code. Available at: www.codeproject.com/vb/net/Refactoring_elixir.asp.

[3] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.
[4] L. Briand, C. Bunse and J. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Trans. on Soft Eng, 27(6), 2001, pages 513—530.
[5] M. Bruntink and A. van Deursen. An empirical study into class testability. Journal of Systems and Software, 2006 (to appear).
[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. Proc 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 et al., (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 Int. Symp on Software Metrics, Chicago, USA, 2004, pages 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 OO 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 Trans. on Soft. Eng. 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 Soft 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, Soft. Testing, McGraw-Hill, 1994.
[26] D. Saff, S. Artzi, J. Perkins and D. Ernst. Automatic test factoring for Java. Proceedings 21st Annual Int. Conf. on Automated Soft Engineering, Long Beach, USA, Nov. 9-11, 2005, pp. 114-123.