Refactoring test suites versus test behaviour - a TTCN-3 perspective

Steve Counsell and Rob M. Hierons
Department of Information Systems and Computing
Brunel University
Uxbridge
+44 (0)1895 266740
{Steve.Counsell, Rob.Hierons}@brunel.ac.uk

ABSTRACT

As a software engineering discipline, refactoring offers the opportunity for reversal of software 'decay' and preservation of a level of software quality. In a recent paper by Zeiss et al. [23], a set of fifteen refactorings were found applicable to Testing and Test Control Notation (TTCN-3) test behaviour and a set of thirteen refactorings to improving the overall structure of a TTCN-3 test suite. All twenty-eight refactorings were taken from the set of seventy-two described in the seminal text by Fowler [10]. An important issue with any refactoring is the testing effort required during implementation of its mechanics. In this paper, we explore the trade-offs between, and the contrasting characteristics of, the two TTCN-3 sets of refactorings from a refactoring mechanics perspective. Firstly, we use a meta-analysis of the same twenty-eight refactorings based on a dependency matrix developed through scrutiny of the mechanics of all seventy-two refactorings in [10] and then an analysis of the refactoring *chains* emerging from each of the same twenty-eight refactorings. Results suggest that there are compelling reasons for avoiding test suite structure refactorings when the dependencies and chains of the test suite refactorings are considered. Refactoring test behaviour potentially offers a far simpler, less demanding set of tasks required of the developer both from a re-testing and dependency viewpoint.

Keywords: Refactoring, Test Suite, Test Behaviour, TTCN-3.

SOQUA'07, September 3-4, 2007, Dubrovnik, Croatia
Copyright 2007 ACM ISBN 978-1-59593-724-7/07/09...$5.00

1. INTRODUCTION

As a software engineering discipline, *refactoring* [10] has grown in prominence over recent years [3, 7, 9, 10, 11, 12, 17, 19]. Refactoring is defined as a change made to software in order to improve its structure without necessarily changing the semantics of the program. In theory, the subsequent improvement in code comprehensibility makes the software easier to maintain and refactoring can provide both short-term and long-term benefits. In fact, Fowler [10] suggests that the process of refactoring is the reversal of evolutionary decay, the preservation of sound software structure and hence, any refactoring effort is thus worthwhile. In the same text by Fowler, the mechanics of seventy-two refactorings are described on a *step-by-step* basis, with a requirement on the part of the developer to test at each logical point in those mechanics. A significant amount of research has been carried out into the practicalities of refactoring both proprietary and Open-Source Software (OSS) software and testing of OO software [1, 4, 8, 15, 16, 20]. Very little research, however has addressed the same issues with respect to refactoring test suites themselves and the subsequent testing requirements that ensures the test suite has been refactored correctly. (It is often the case that a test suite is larger than the code it is testing.) In this paper, we examine two sets of refactorings. The first is set of fifteen test *behaviour* refactorings and the second a set of thirteen test suite *structure* refactorings adapted to Testing and Test Control Notation (TTCN-3) constructs [22]. TTCN-3 is an industry standard notation for controlling the testing process. Many of its constructs explicitly allude to, or can be adapted to, the features of a typical OO language.

We use a meta-analysis of the twenty-eight refactorings based on a dependency matrix developed through scrutiny of the mechanics of all seventy-two refactorings. The dependency matrix shows, for each of the twenty-eight refactorings, the number of other, dependent refactorings. We then analyse the chains generated by each of the twenty-eight refactorings due to their inherent dependencies. Results indicate that there are good reasons for a developer to avoid test suite refactorings in favour of refactoring test behaviour; the latter offers a relatively simple set of requirements both from a re-testing and dependency viewpoint. From a wider perspective, refactoring has a direct bearing on potential software quality through improved comprehensibility, subsequent maintainability and the influence this may have on fault incidence. The paper is structured as follows: in the following section, we describe the motivation for the research and related work. In Section 3, we describe the set of twenty-eight TTCN-3-based refactorings and an analysis of each of those twenty-eight refactorings with respect to the refactorings they each use (Section 4). In Section 5, we extend the analysis to take account of *chains* (i.e., sequences) of refactorings generated by each of the same set of refactorings. We discuss some of the issues arising from our research in Section 6, before concluding in Section 7.

2. MOTIVATION/RELATED WORK

The research described in this paper is motivated by three factors. First, the size of test suites is becoming increasingly difficult to manage and control. Refactoring techniques offer opportunities and potential for reducing complexity in those test suites. Second, while the TTCN-3 language is a recognised standard for testing systems [22], the potential for applying what are, effectively, Java-based refactorings to test suites has not been the source of much investigation. Third, while it is entirely feasible to propose refactorings for both test behaviour *and* test suite structure, the two sets are likely to have their own distinct and interesting properties; a developer considering a refactoring or set of refactorings should thus be aware of those properties, since they may influence the decision on which refactorings to apply.

The work in this paper builds on other previous research by the authors where we investigated the link between refactoring and testing. In [5], we adapted the testing taxonomy proposed by Van Deursen & Moonen (VD&M) based on the post-refactoring repeatability of tests [8]. 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 [10] where minimum disruption to testing effort was the goal. Herein, we specifically explore the nature of the twenty-eight test-based refactorings proposed by Zeiss et al., [23] in greater detail.

In terms of other related work, recent work by Advani et al., [1] describes the results of an empirical study of the key trends across multiple versions of the same Java OSS. 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'. Results showed the most common refactorings of the fifteen coined a 'Gang of Six', to be generally those with a high in-degree and low out-degree (arcs entering and emerging, respectively) when mapped on a dependency graph; the same refactorings also featured strongly in the remedying of bad code smells. Surprisingly, inheritance and encapsulation-based refactorings were found to have been applied relatively infrequently - we offered explanations for why this may be the case. The paper thus identified 'core' refactorings central to many of the changes made by developers on open-source systems. A study of the trends in changes, categorised according to refactorings was also undertaken in [6] and an investigation of change metrics as a basis of refactorings in [7]. A full survey of recent refactoring work can be found in [14]. Finally, we note that research in which the twenty-eight refactorings were proposed [23] represents just one of a number of research works by the same authors, primarily looking into tool-based support for refactoring test suites and metrics for analyzing TTCN-3 specifications [2]

3. THE TTCN-3 REFACTORINGS

3.1 Test Behaviour

In Zeiss et al., [23] a set of fifteen refactorings applicable for refactoring TTCN-3 test behaviour were described. The set of fifteen refactorings were all taken from Fowler's text [10] and adapted by Zeiss et al., to their TTCN-3 equivalents with re-interpretation where necessary; henceforward, we will refer to these refactorings as Test Behaviour (TB) refactorings. The fifteen refactorings are shown in Table 1, together with a description of the purpose of each. One key difference between TTCN-3 notation and object-oriented code is the absence in the former of any notion of a method (in TTCN-3 it becomes a function). For that reason, two of the fifteen refactorings in Table 1 enclosed in square brackets (i.e., 4. Extract Method and 7. Inline Method) have been modified to become 'Extract Function' and 'Inline Function', respectively. The predicate on which our research rests is that Java-based refactorings drawn from Fowler have a strong correspondence with the TTCN-3 notation and thus require only minor adaptation. The refactorings in Table 1 demonstrate that only minor changes are necessary to make that transition in these cases.

Table 1. The Test Behaviour (TB) refactorings of Zeiss et al., [23].

Refactoring	Description
1. Consolidate Conditional Expression	A sequence of conditional tests with the same result is combined into a single conditional expression.

2. Consolidate Duplicate Conditional Fragments	A fragment of code appearing in all branches of a conditional are extracted to a single place.
3. Decompose Conditional	Each part of a complicated conditional is split into extracted methods.
4. Extract Function [Extract Method]	A code fragment can grouped together; turn that fragment into a function [method] whose name explains its purpose.
5. Introduce Assertion	A section of code assumes something about the state of the program; the assumption is made explicit with an assertion.
6. Introduce Explaining Variable	A complicated expression is decomposed by using temporary variables for parts of the expression.
7. Inline Function [Inline Method]	The body of a function [method], whose name and purpose is obvious is moved into the body of its caller.
8. Inline Temp	A temporary variable is replaced with a corresponding expression.
9. Remove Assignment to Parameters	A temporary variable is used to replace an assignment to a parameter.
10. Remove Control Flag	A variable acting as a control flag is replaced with a 'break' or 'return'.
11. Replace Nested Conditional with Guard Clauses	An unclear nested 'if' statement is replaced with a set of guard clauses for the special cases.
12. Replace Temp with Query	A temporary variable holds the result of an expression; the expression is extracted into a method.
13. Separate Query from Modifier	A method that a) returns a value and b) changes the state of an object is replaced by separate methods for a) and b)
14. Split Temporary Variable	A temporary variable is assigned to more than once (and is not a loop variable); a separate variable is used for each assignment
15. Substitute Algorithm	An algorithm is replaced by one which is simpler.

Noticeable from Table 1 is the strong influence of simple variable, expression and condition manipulation, as we would expect for test behaviour refactorings; it is this simplicity that explains why only two of the fifteen refactorings need to be re-interpreted for TTCN-3.

3.2 Test Suite Structure

In the same paper by Zeiss et al., [23] a set of thirteen refactorings applicable to test suite structures was described. Again, the set of thirteen refactorings was taken from Fowler's text [10] and adapted by Zeiss et al., to TTCN-3 where necessary by re-interpreting the meaning of the refactoring; henceforward, we refer to these refactorings as Test Suite Structure (TSS) refactorings. In TTCN-3, there is no concept of a *class* – the term *component* was used by Zeiss et al., in its place. Equally, the notion of extracting a *subclass* or *superclass* does not make sense in TTCN-3 even though inheritance is a feature of TTCN-3. The terms *Extended Component* and *Parent Component*, respectively are used instead. Finally, a *field* in the Java-based refactorings of Fowler [10] has no meaning in TTCN-3 and is thus replaced by one of: *Port*, *Variable*, *Constant* or *Timer* all four of which are potentially usable in TTCN-3. The minor changes required to the refactorings in Table 2 illustrate the applicability of Java-based refactorings to a TTCN-3 context.

Table 2. The Test Suite Structure (TSS) refactorings of Zeiss et al., [23].

Refactoring	Description
1. Add Parameter	A function [method] needs more information from its caller. A parameter is used to pass that information.
2. Extract Extended Component [Extract Subclass]	A component [class] has features that are only used in some instances. A component [subclass] is created for that subset of features.
3. Extract Parent Component [Extract Superclass]	Two components [classes] have similar features. A parent component [superclass] is created and the common features are moved to that parent component [superclass].
4. Introduce Local Port/Variable/Constant/Timer	A server component [class] needs additional functions [methods], but can't be modified. A new component

[Introduce Local Extension]	[class] is created and becomes an extended component [subclass] of the original class.
5. Introduce Record Type Parameter [Introduce Parameter Object]	A group of parameters that naturally go together are replaced with a record type [object].
6. Parameterise Test Case/Function/Alt Step [Parameterise Method]	Several functions [methods] do the same thing with different values; a single function [method] is created to handle both values.
7. Pull Up Port/Variable/Constant/Timer [Pull Up Field]	Two extended components [subclasses] have the same field. The Port/Variable/Constant/Timer [field] in question is moved to the parent component [superclass].
8. Push Down Port/Variable/Constant/Timer [Push Down Field]	A Port/Variable/Constant/Timer [field] is used only by some components [subclasses]. The field is moved to those components [subclasses].
9. Replace Magic Number with Symbolic Constant	A constant replaces a literal value.
10. Remove Parameter	A parameter is no longer used by a function [method] body and is removed.
11. Rename [Rename Method]	A function [method] is renamed to reflect its purpose in a clearer way.
12. Replace Parameter with Explicit Functions [Replace Parameter with Explicit Methods]	A function [method] runs different code depending on the values of an enumerated parameter; a separate function [method] is created for each value of the parameter.
13. Replace Parameter with Function	A component [object] invokes a function [method] and passes the results as a parameter to a receiving component [object]; the receiver should invoke the function [method].

In contrast to the two out of fifteen TB refactorings that need to re-interpreted to TTCN-3 semantics, ten of the thirteen TSS refactorings require TTCN-3 re-interpretation. This suggests that, on the face of it, while adaptation is not necessarily a problem, the TSS refactorings are less easily adapted to TTCN-3 than the TB refactorings. Many of the TSS refactorings are structural in nature; for example, refactorings 2, 3, 7 and 8 from Table 2 all require modification and appreciation of the structure of the inheritance hierarchy in both the Java and TTCN-3 sense; it is largely differences in the semantics of these building blocks of Java and TTCN-3 that accounts for the extra adaptation effort.

4. DEPENDENCY ANALYSIS

As part of our refactoring analysis and to inform our understanding of the twenty-eight refactorings, we developed a dependency matrix showing all seventy-two of Fowler's refactorings and how they were inter-related. For each refactoring let's say, X, the out-degree taken from the graph (i.e., the number of directed arcs emerging from that node) illustrated the refactorings that were *used by* X. The matrix was developed through close examination and scrutiny of the seventy-two refactorings in Fowler's text. We would want a refactoring to have a zero out-degree (i.e., because that would indicate that the refactoring does not require the use of any other refactorings as part of its mechanics). In fact, the lower the out-degree, the lower the dependency of that refactoring on other refactorings and, in theory, the easier the refactoring is to complete.

4.1 In-degree and out-degree analysis

Figure 1 shows the out-degree values for the fifteen TB refactorings. The maximum value among the fifteen refactorings is for refactoring 4, 'Extract Function'. This refactoring uses 'Remove Assignments to Parameters', 'Replace Method with Method Object', 'Replace Temp with Query' and 'Split Temporary Variable' refactorings. Interestingly, three of these refactorings appear in the same set of TB refactorings. The refactoring that uses three other refactorings is 'Replace Temp with Query', which uses 'Inline Temp', 'Split Query from Modifier' and 'Split Temporary Variable', all three of which are in the same set of TB refactorings. Seven of the fifteen refactorings use *zero* other refactorings as part of their stated mechanics (i.e., 2, 6, 7, 8, 9, 14 and 15) and five of the fifteen refactorings use only one other refactoring as part of their mechanics (i.e., 1, 3, 5, 10 and 13). We assume that each of these associated refactorings can be easily adapted to the TTCN-3 notation. For example, of the four refactorings used by 'Extract Function', only 'Replace Method with Method Object' needs modification to become 'Replace Function with Record Type'.

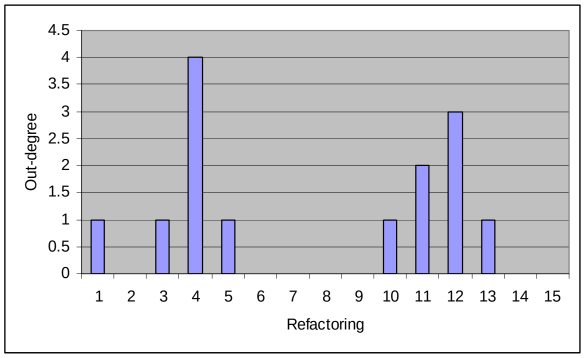

Figure 1. Out-degree for the fifteen TB refactorings.

To complete the picture, Table 3 shows the refactorings that each of the fifteen TB refactorings use as part of those mechanics. (This information was extracted directly from the dependency matrix.) Interestingly, and a result we did not anticipate is the high number of inter-relationships (i.e., overlap) between the TB refactorings. Table 3 shows (in bold font), the refactorings used by TB refactorings that are actually TB refactorings themselves. Only two of the 14 refactorings are drawn from outside the TB set, namely 'Replace Method with Method Object' and 'Replace Magic Number with Symbolic Constant'.

Table 3. The refactorings that the fifteen TB refactorings *use*

Refactoring X	Refactorings that X uses
1. Consolidate Conditional Expression	**Extract Method**
3. Decompose Conditional	**Replace Nested Conditional with Guard Clauses**
4. Extract Function [Extract Method]	**Remove Assignments to Parameters**, Replace Method with Method Object, **Replace Temp with Query, Split Temporary Variable.**
5. Introduce Assertion	**Extract Method.**
10. Remove Control Flag	**Separate Query From Modifier**
11. Replace Nested Conditional with Guard Clauses	**Consolidate Conditional Expression,** Replace Magic Number with Symbolic Constant
12. Replace Temp with Query	**Inline Temp, Separate Query from Modifier, Split Temporary Variable.**
13. Separate Query from Modifier	**Substitute Algorithm**

Figure 2 shows the out-degree values for the thirteen TSS refactorings.

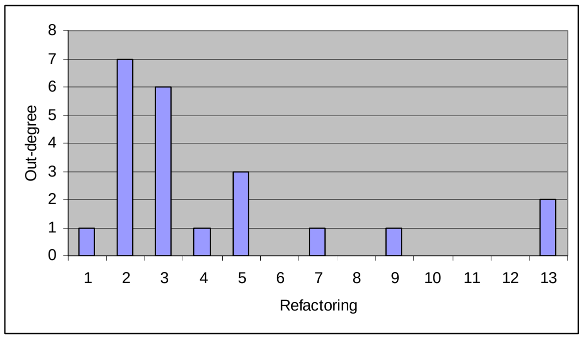

Figure 2. Out-degree for the thirteen TSS Refactorings.

The refactoring with the highest out-degree is for refactoring 2, 'Extract Extended Component' with an out-degree of 7; the refactoring with an out degree of 6 is 'Extract Parent Component'. Five of the thirteen refactorings use no other refactorings as part of their mechanics (i.e., 6, 8, 10, 11 and 12). To complete the picture, Table 4 shows the refactorings that each of the thirteen TSS refactorings use as part of their mechanics.

Table 4 also shows (in bold font), the other required TSS refactorings that belong in the set of TSS refactorings. Seventeen of the twenty-one refactorings are drawn from outside the TSS set; the trend in Table 4 is thus in complete contrast to the TB refactorings where a strong inter-dependence was noted. This result suggests that if a developer wants to apply one of the TSS refactorings, they would, generally speaking, first need to understand the mechanics of refactorings outside the core set of applicable TSS refactorings. On the other hand, the TB refactorings tend to use refactorings taken from the same set. The latter feature has a major advantage: if developers are continually applying refactorings from the same set of refactorings (in this case the TB set), they are likely to become more adept at using those refactorings and, consequently, will become trivial to apply.

Table 4. The refactorings that the thirteen TSS refactorings *use*

Refactoring X	Refactorings that X uses
1. Add Parameter	Introduce Parameter Object.
2. Extract Extended Component [Extract Subclass]	Move Method, Push Down Method, **Push Down Field**, **Rename Method**, Replace Conditional with Polymorphism, Replace Constructor with Factory Method, Self Encapsulate Field.
3. Extract Parent Component [Extract Superclass]	Form Template Method, Pull Up Constructor Body, **Pull Up Field**, Pull Up Method, **Rename Method**, Substitute Algorithm.
4. Introduce Local Port/Variable/Constant/Timer/ [Introduce Local Extension]	Move Method
5. Introduce Record Type Parameter [Introduce Parameter Object]	**Add Parameter**, Extract Method, Move Method.
7. Pull Up Port/Variable/Constant/Timer [Pull Up Field]	Self Encapsulate Field.
9. Replace Magic Number with Symbolic Constant	Replace Type Code with Class.
13. Replace Parameter with Function [Method]	Hide Method, Remove Control Flag

5. REFACTORING CHAINS

There is far more serious implication for the TB and, particularly the TSS refactorings, relating to the possibility that a *chain* of required refactorings (i.e. a sequence of refactorings) is induced by each refactoring. Chain information can be extracted from our dependency matrix by following for, let's say, refactoring X, the links of every refactoring that X uses. The basis on which chains rest is that each of the refactorings on the right hand side column of Tables 3 and 4 may, in turn, require the use of many other refactorings. More formally, we say that a refactoring X has a chain of length *n* if, for that refactoring, there are *n-1* subsequently refactorings that need to be undertaken. A chain terminates when no more refactorings in that sequence can be found (i.e., a *terminal* refactoring, requiring the use of zero other refactorings, is reached for every refactoring in the chain). A refactoring X can thus have many chains of varying lengths, reflecting the different combinations of refactorings it uses. A refactoring with relatively large number of chains will have significant implications for the testing effort during refactoring. We note *en passant* that the notion of a refactoring inducing many other refactorings is not a new one; it is seen as one of the current challenges facing the refactoring community [13, 18].

Figure 3 shows the number of chains for the two sets of refactorings (the TB refactorings are denoted by the left-most, lighter bars). For example, refactoring 1 in the TB set of refactorings is 'Consolidate Conditional Expression' and gives rise to seven chains through analysis of the dependency matrix; refactoring 1 in the TS refactorings is 'Add Parameter' and gives rise to eight chains. Of particular note in Figure 3 is the relatively high number of chains given by TSS refactorings 2 and 3. These two refactorings are 'Extract Subclass' (cf. Extract Extended Component) and 'Extract Superclass' (cf. Extract Parent Component) giving rise to 18 chains apiece. A developer would want to avoid these latter two refactorings and this stance is shared with the empirical profile of a previous study [1], where these two refactorings were shown to be undertaken very sparingly. In other words, refactorings with a strong structural basis (e.g., inheritance related refactorings) need to be treated with care.

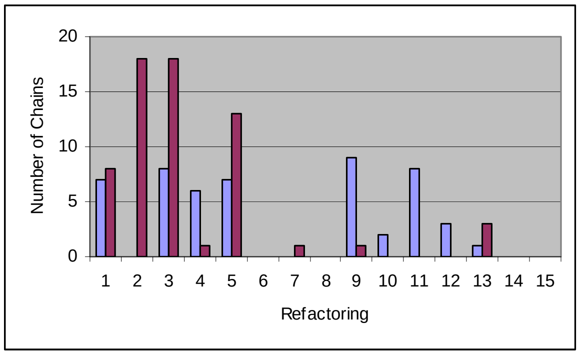

Figure 3. Numbers of refactoring chains induced by TB and TSS refactorings.

The message for any developer is clear: if you intend refactoring the structure of a test suite, then the likelihood is that you will have to employ a wider variety of refactorings than those directly applicable to test suites. The degree of re-interpretation necessary for TTCN-3 is significantly less for test behaviour refactorings. In addition, there are certain refactorings in the set of TSS refactorings that should be treated with extreme caution because of the potentially long refactoring chains they induce.

6. DISCUSSION

There are many issues that arise as a result of the analysis described. Firstly, for many of the seventy-two refactorings described in [10] the refactoring mechanics prescribe that a particular refactoring *must* use refactoring X in order to complete a refactoring. For example, the dependency matrix shows that the 'Introduce Parameter Object' requires the use of the 'Add Parameter' refactoring in a 'must use' relationship. Just as 'must use' relationships specify refactorings that must be undertaken to facilitate another refactoring, some refactorings 'may' require the use of other refactorings if the conditions hold during refactoring. For example, the Encapsulate Field refactoring 'may' use Move Method after it has been completed depending on whether the developer decides it is worthwhile and applicable (again, this information was obtained from the dependency matrix). We could thus refine our analysis by modifying the dependency matrix to include an indication of whether refactorings were used in 'must' or 'may' relationship with other refactorings. A refactoring with chains incorporating only 'may use' relationships would be far more preferable to a refactoring with only 'must use' relationships in its chains. This remains a topic of future research.

Secondly, we have analyzed the mechanics of twenty-eight refactorings adapted to TTCN-3 on the basis that there is a mapping between the Java mechanics and those of TTCN-3. The study could be criticized because the mechanics of the Java refactorings would not apply in a TTCN-3 context. However, on the basis that we are able to find analogies for OO: inheritance, methods and fields in TTCN-3 (along the lines that Zeiss et al. [23] describe) the mechanics of each of the twenty-eight refactorings (including refactorings that those twenty-eight refactorings use) can also be modeled in a TTCN-3 sense. Equally, for the remaining refactorings specified by Fowler [10] and used in Tables 3 and 4, we were able to find analogies in the TTCN-3 sense. Finally, we have assumed that the developer has a choice as to which refactorngs they may want to undertake and is able to opt for test behavior refactorings as opposed to test suite structure refactorings. Ultimately, the extent of code *smell* [10] may be the deciding factor. We do accept that other factors may influence the decision as to which refactorings are undertaken. We have also made no assumption about the software tool support available and the speed it offers for assisting the developer; this could be a confounding factor for our analysis. However, considering carefully the prior analysis, even with the use of a tool, the time to undertake a TSS refactoring is still likely to be correspondingly greater than for a TB refactoring.

The quality of software has always been dependent on the quality of the testing; nowadays, industry is facing an uphill task to cope with the size of test suites and their associated behaviour. The key research issue that this paper raises is the mapping from Fowler's refactorings to those of TTCN-3. The fact that Zeiss et al., [23] adopted twenty-eight of those refactorings suggests a strong bond. A wide range of research problems thus become apparent. For example, what quantifiable benefits and trade-offs are there to refactoring test suites (against test behaviour). Equally, some chains may be less harmful in terms of their required effort than other identically sized chains. These are just two challenges that emerge from the analysis herein.

7. CONCLUSIONS

In this paper, we have compared two sets of refactorngs; the first, a set of fifteen test behaviour (TB) refactorings and the second, a set of thirteen test suite structure (TSS) refactorings proposed by Zeiss [23]. Where necessary these twenty-eight refactorings were adapted to the semantics of TTCN-3 [22]. We demonstrated three compelling reasons why a developer would want to be careful about choosing test suite refactorings in favour of test behaviour refactorings, related to the out-degree of TSS refactorings when mapped on a dependency matrix and the refactoring chains induced by TSS refactorings. Test suite refactorings should be given very careful consideration before being undertaken. Refactoring test behaviour on the other hand require a simpler, less demanding set of tasks required of the developer from both a re-testing and dependency viewpoint. The research described herein represents the start of an analysis into the potential for refactoring test suites. Notwithstanding the analysis in this paper, there are still many issues and challenges remaining in this growing and important area.

8. 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 Symposium on Applied Computing, Dijon, France, April 2006.

[2] P. Baker, D. Evans, J. Grabowski, H. Neukirchen and B. Zeiss. TRex - The Refactoring and Metrics Tool for TTCN-3 Test Specifications. Proceedings of the Testing: Academic and Industrial Conference on Practice (TAIC PART), Windsor, UK, August 2006, pages 90-94, IEEE Computer Society Press.

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

[4] M. Bruntink and A. van Deursen. An empirical study into class testability. Journal of Systems and Software, 2006.

[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 Testing: Academic and Industrial Conference (TAIC PART), Windsor, UK, August 2006, pages 181-190. IEEE Computer Society Press.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

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

[17] R. Najjar, S. Counsell and G. Loizou. Encapsulation and the vagaries of a simple refactoring: an empirical study. Proceedings Int. Conference on Software Systems Engineering and its Applications, Paris, France, Dec. 2005.

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

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

[20] 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.

[21] 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.

[22] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler and S. Schulz. An Introduction to TTCN-3, Wiley, 2005.

[23] B. Zeiss, H. Neukirchen, J. Grabowski, D. Evans and P. Baker: Refactoring and Metrics for TTCN-3 Test Suites. 5th Workshop on System Analysis and Modelling (SAM), Kaiserslautern, Germany, May 2006, pages 148-165.