Automated Refactoring for Testability

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin
Ireland
Email: mel.ocinneide@ucd.ie

Dermot Boyle
Microsoft Labs
Dublin
Ireland
Email: dboyle@microsoft.com

Iman Hemati Moghadam
School of Computer Science and Informatics
University College Dublin
Ireland
Email: Iman.Hemati-Moghadam@ucdconnect.ie

***Abstract*—Current software practice places a strong emphasis on unit testing, to the extent that the amount of test code produced on a project can exceed the amount of actual application code required. This illustrates the importance of *testability* as a feature of software. In this paper we investigate whether it is possible to improve a program's testability using an automated refactoring approach. We conduct a quasi-experiment where we create a small application that scores poorly using a proven cohesion metric, LSCC. Using our automated refactoring platform, Code-Imp, this application is automatically refactored using the LSCC metric to guide the search for better solutions. To evaluate the results, a number of industrial software engineers were asked to write test cases for the application both before and after refactoring and compare the relative difficulty involved. The results were interesting though inconclusive, and suggest that further work is required.**

I. Introduction

Testability is a highly desirable external attribute of software. Software that ranks highly on testability measures can be expected to be easier to write test cases for, enable more concise and readable test cases and be less likely to contain errors undetected by the test cases. Testability is also a very broad concept, incorporating issues from architecture through to implementation. For example, at an architectural level, user interface software must be separated from business logic software, otherwise the business logic cannot be tested without being entangled with interface issues. At the other extreme, at an implementation level, nested conditional statements can be converted to a single compound conditional to reduce the number of paths through the code and hence reduce testing effort [23].

In this work, we are concerned with *design* issues that affect testability. At a design level, the key software quality metrics are based on coupling and cohesion. Studies by Bruntink and van Deursen [6] and more recently by Badri, Badri and Toure [4] have shown that cohesion metrics are good predictors for testability, where testability is measured inversely by the number of lines of test code and the number of `assert` statements in the test code. These studies prompted us to consider if it is possible to improve the testability of a program by improving its cohesion measure through automated refactoring, and this is the research question addressed in this paper.

A multitude of cohesion metrics are described in the literature. We decided to use the LSCC metric (Low-level design Similarity-based Class Cohesion) which was introduced by Al Dallal and Briand [10]. We choose this metric for a number of reasons:

- LSCC is a low-level design metric, suitable for applying to source code.
- It has been evaluated on industrial examples and shown to measure a dimension of cohesion not detected by other cohesion metrics.
- The authors demonstrate its usefulness in guiding class refactoring.

This leads to the question of how we can improve the LSCC measure for a program. In previous work, we developed an automated refactoring platform, Code-Imp, that can be used to improve the design of a program according to certain metrics [17]–[19]. In this paper, we use Code-Imp to refactor the design of a program to improve its LSCC score. The Code-Imp platform is described in more detail in section III.

Our approach in this work is to first develop a small Java application that has severe cohesion problems. We refactor this program using Code-Imp in order to improve its design according to the LSCC cohesion metric. We then invited a number of industrial software engineers to write test cases for the both versions of the program, before and after refactoring, and to compare the difficulty in writing the test cases. If writing test cases for the refactored version of the program proves significantly easier, then there is indicative evidence that automated refactoring can indeed improve testability.

The remainder of the paper is structured as follows. In section II we describe related work. Our automated refactoring platform, Code-Imp, is the topic of section III, and in section IV we present in detail our experiment to test if automated refactoring can improve testability. The results of this experiment are presented and evaluated in V. Possible future work is outlined in section VI and we present our conclusions in section VII.

II. Related Work

Many structural cohesion metrics have been proposed for object-oriented programs both at design and implementation levels, e.g. Tight Class Cohesion (TCC) of Bieman and Kang [5], Lack of Cohesion between Methods (LCOM) of Chidamber and Kemerer [8], Normalised Hamming Distance (NHD) of Counsell et al. [9], Similarity-based Class Cohesion (SCC) of Briand and Al Dallal [1]. Cohesion is a difficult property to measure since structural measurements do not

directly capture semantic cohesion. Hence the debate as to which approach to cohesion is best is ongoing and new cohesion metrics are still appearing [1], [4], [10].

Bruntink and van Deursen [6] studied five open source Java projects that had unit tests to determine if a correlation could be found between various structural metrics and unit test metrics. The unit test metrics used were lines of test code and number of `assert` statements, which are proposed as inverse measures of testability. They found significant correlation in the case of the coupling metrics used (fan-out, lines of code per class and Response For a Class (RFC) [8]) and the unit test metrics. In the case of the single cohesion metric they studied (LCOM), some correlation was found, though this appeared to be dependent on the application. Badri, Badri and Toure performed a similar study more recently [4]. They used two open source Java systems and tried to find a correlation between a lack of cohesion and low testability characteristics. The cohesion metrics they used were LCOM, LCOM* and LC_D and their notion of test case quality was the same as in the work of Bruntink and van Deursen above. They found a clear correlation between cohesion and testability and this has motivated our use of cohesion metrics in this work.

Alshayeb investigated the impact refactoring has on external software quality attributes, including testability [2]. In his experiments he found that the refactorings performed by students had a mixed impact on the testability of the software. Rather than try to measure testability directly, he measured the metrics that Bruntink and van Deursen [6] had found to be good predictors of testability. In general, his work failed to show that refactoring improved any of the external quality attributes studied, namely adaptability, maintainability, understandability, reusability, and testability[1].

Search-Based Refactoring is fully automated refactoring driven by metaheuristic search and guided by software quality metrics. It was introduced in our previous work by O'Keeffe and Ó Cinnéide [20], where it was used with some success to automate the improvement of software design [18], [19]. Seng et al [22] developed a search-cased refactoring approach using a genetic algorithm and a novel evaluation function based largely on the Chidamber and Kemerer metrics suite [8]. They used this to successfully reposition displaced methods in the class structure of HotDraw. Harman and Tratt [15] demonstrated the benefit of using pareto optimality in search-based refactoring. Otero et al [21] use search-based refactoring to refactor a program as it is being evolved using genetic programming in an attempt to find a different design which may admit a useful transformation as part of the genetic programming algorithm. Jensen and Cheng [16] use genetic programming to drive a search-based refactoring process that aims to introduce design patterns.

[1]A similarly curious result was found by Chatzigeorgiu who studied the evolution of code smells in open source software and found that very few of them were actually removed by refactoring [7].

III. The Code-Imp Refactoring Platform

Code-Imp is an extensible platform for automated refactoring that we have previously used for automated design improvement [18], [19]. It is focussed on design-level refactorings such as moving methods around the class hierarchy, splitting classes and changing inheritance and delegation relationships. It does not attempt to split or merge methods. Code-Imp was recently reengineered to use the RECODER platform [13] and now supports Java 6. It currently implements the following refactorings:

Method-level Refactorings:

- *Push Down Method*: Moves a method from some class to those subclasses that require it.
- *Pull Up Method*: Moves a method from some class(es) to their immediate superclass.
- *Increase/Decrease Method Security*: Changes the security of a method by one level, e.g. private to protected or public to default.

Field-level Refactorings:

- *Push Down Field*: Moves a field from some class to those subclasses that require it.
- *Pull Up Field*: Moves a field from some class(es) to their immediate superclass.
- *Increase/Decrease Field Security*: Changes the security of a field by one level, e.g., private to protected or public to default.

Class-level Refactorings:

- *Extract Hierarchy*: Adds a new subclass to a non-leaf class C in an inheritance hierarchy. A subset of the subclasses of C will inherit from the new class.
- *Collapse Hierarchy*: Removes a non-leaf class from an inheritance hierarchy.
- *Make Superclass Abstract*: Declares a constructorless class explicitly abstract.
- *Make Superclass Concrete*: Removes the explicit abstract declaration of an abstract class without abstract methods.
- *Replace Inheritance with Delegation*: Replaces an inheritance relationship between two classes with a delegation relationship; the former subclass will have a field of the type of the former superclass.
- *Replace Delegation with Inheritance*: Replaces a delegation relationship between two classes with an inheritance relationship; the delegating class becomes a subclass of the former delegate class.

Code-Imp drives the refactoring process using one of a number of metaheuristic search techniques. In this work, straightforward first-ascent hill climbing was used, using the LSCC metric as the fitness function. So at each stage in the refactoring process, a random refactoring is chosen to be applied. If it improves LSCC, the refactoring is accepted and the search for another refactoring restarts with the new program. If the refactoring fails to improve LSCC, it is rejected and another refactoring is randomly chosen. The process stops when no refactoring can be found that improves LSCC.

IV. Experimental Design

We wish to test the hypothesis that automated refactoring can improve the testability of a program. Our approach is to develop a Java application that has serious cohesion problems (the "before" version, termed version A). This application is then automatically refactored using the Code-Imp platform guided by the LSCC cohesion metric to produce the "after" version, termed version B. Both these versions exhibit the same external functionality, but have distinctly different internal designs. We then invited a number of industrial software engineers to write a series of test cases for both versions, and to assess their relative difficulty. Given the simplicity of the test cases in these examples, it was decided that an objective measure of test case quality, such as code coverage or mutation testing, would not yield much differentiation.

This quasi-experiment[2] is detailed in the following sections.

A. Sample Programs

For this proof-of-concept experiment, we created a small Java application of 14 classes that models people in various roles (student, teacher, manager etc.). After building the application, we refactored it by hand with the goal of reducing its cohesion. This was mainly achieved by moving methods and fields from their natural class to another related, but inappropriate, class. This initial uncohesive program is termed version A.

We ran Code-Imp using version A as input six times. First-ascent hill-climbing has a random element, so different results were achieved each time. We selected the run that produced the longest sequence of refactorings (26) to use in the experiment. This choice was made purely on the basis of the length of the refactoring sequence and did not involve examining the final code of the refactored program. Figure 1 presents an overview of the refactorings that were performed. The design was changed quite radically – seven new classes were added (by *Extract Hierarchy*), one interface was added (by *Replace Inheritance with Delegation*) and a total of 13 method and field displacements occurred[3]. The LSCC cohesion metric was improved from an initial value of 0.042 in version A to a final value of 0.088 in Version B. It is very possible that the score for some other metrics decreased. However, as described in section IV-B, the testability exercises are designed to focus on parts of the program that appeared to have improved in the refactored version.

On inspection, the design of version B appeared indeed to be an improvement over the design of version A. In previous experiments with Code-Imp we found that certain metrics can lead the refactoring process to ruin the program design [19], so it was interesting to note that, in this example at least, the LSCC metric gave rise to a refactoring sequence that led to a stable result.

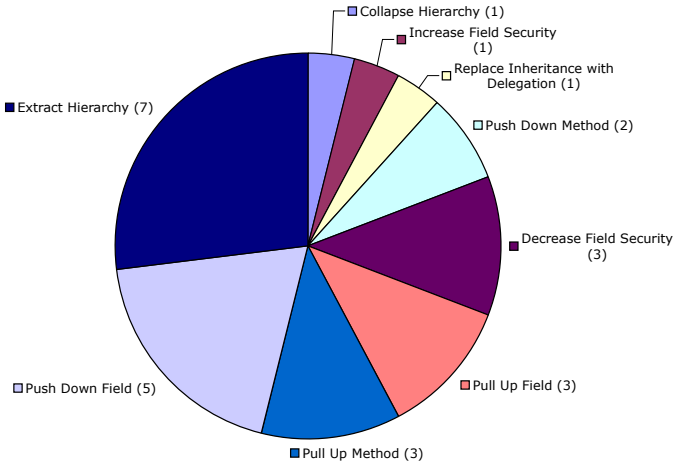

Fig. 1. Breakdown of the 26 refactorings applied to version A to produce version B.

B. Exercises

Our hypothesis can now be expressed that version B of the program is easier to test than version A. To test this, we invited 10 experienced software engineers to compare version A of the Java application that had cohesion issues with the refactored version B. Even for a small application like this, making an overall assessment of testability is difficult, vague and time-consuming. Hence we created six exercises related to testability that were to be performed on both versions. The participants were then asked to assess the relative difficulty of the exercise for version A and version B on a 7-point Likert scale as follows:

1) Version A is much easier to test.
2) Version A is moderately easier to test.
3) Version A is slightly easier to test.
4) Both are the same / I have no opinion.
5) Version B is slightly easier to test.
6) Version B is moderately easier to test.
7) Version B is much easier to test.

In creating the exercises, we deliberately chose parts of the application where the A version had a design problem that was resolved in the B version, and where we felt that this difference would make writing test cases easier. The full survey and results can be obtained by emailing the authors (MÓC); here we summarise the nature of each exercise we created:

- **Exercise 1**: The class `Industrialist` in both versions provides the functionality to set the industrialist's name. In version A this method is inherited from a superclass while in version B it is implemented by delegation to another class. This exercise involved writing a test case for this method for both versions.

[2]In a formal experiment, participants would be randomly assigned to the before and after programs. Given the small numbers involved in this experiment, each participant did the test cases for both before and after versions.

[3]To aid program comprehension, the new classes and interface added were given sensible names rather than the names generated by Code-Imp. This is the one aspect of our approach that is not fully automatable. Automated generation of suitable names for classes and methods is a possibility using a lexical database such as Wordnet [3], [11].

- **Exercise 2**: This exercise involved writing test cases for the constructor of the `Industrialist` class in both versions. As in exercise 1, the essential difference is in testing a class that uses inheritance (version A) versus a class that uses delegation (version B).
- **Exercise 3**: In version A, the functionality to validate input is split between one class which provides the method, and a superclass that stores the fields that the method has to access. In version B, the method and the fields it uses have been moved to the same class. This exercise involved writing a test case for this method for both versions.
- **Exercise 4**: In both versions, a class `Company` provides the functionality to set the company's manager. In version A the manager is stored in `Company` as a field of the class `Person`, and the `setManager` method expects a `Person` object as parameter. In version B the manager is stored as a field of the interface `InterIndustrialistPerson`, and the parameter to `setManager` is also of this interface type. This exercise involved writing a test case for setting a new company manager for both versions.
- **Exercise 5**: Both versions have an identical utility function `getTextValue` that is in the class `Student` in version A and moved to the class `Trainee` in version B. The difference is that in version A `getTextValue` is defined in a complex class of 100 lines of code, while in version B it is the only method in a class of only 20 lines of code. This exercise involved writing a test case for the `getTextValue` method in both versions.
- **Exercise 6**: In version A, the class `Industrialist` is a subclass of `Person`. In Version B, this inheritance relationship does not exist, but `Person` and `Industrialist` both implement the interface `InterIndustrialistPerson`. The exercise is to compare the difficulty in writing test cases for each version.

Exercises 1 and 2 relate to the difference between inheritance and delegation. It is widely accepted that delegation should be preferred over inheritance [12], so we anticipate that the version that uses delegation would prove easier to test. Exercise 3 simply compares writing a test case for a method that uses a field in the same class with one that uses a field defined in the superclass. It would seem obvious that the former design would be easier to test. Exercise 4 examines if it is easier to test a method that uses a parameter of concrete class, or one of a appropriate interface type? We would expect the latter to be the preferred option. Exercise 5 simply compared the difficulty of testing a method when it is surrounded by a lot of 'noise', i.e., situated in a more complicated class. Obviously, one would expect the method in the less noisy context to be easier to test. Exercise 6 was more challenging. It is not recommended to inherit from a concrete superclass as in version A; the preferred solution is for both concrete classes to share a common abstract superclass or interface, as in version B. We anticipated that the latter structure would be preferred, and that some participants would note the possibility of parameterised test cases to lessen the burden for version B.

We are conscious that we have "cherry-picked" these exercises in order to demonstrate the concept of refactoring for testability. This experiment does not preclude the possibility that refactoring to improve LSCC reduced testability in other parts of the program, though we did not see evidence of this. In any case, the results were very surprising, as we see in the next section.

V. Results and Evaluation

The survey above was sent to 14 volunteers on a masters programme in Advanced Software Engineering in University College Dublin. 10 responses were received. Each participant had between 4 and 20 years industrial software engineering experience, of which between 4 and 10 years were spent in development. Each had an average of over 4 years experience of unit testing, which ranged from 0.5 to 12 years.

Participant	Experience (years)	E1	E2	E3	E4	E5	E6	Avg
1	10	4	4	4	3	4	1	3.3
2	20	3	2	4	2	4	3	3.0
3	20	3	4	1	1	4	2	2.5
4	4	5	4	4	4	7	5	4.8
5	5	4	4	4	4	4	4	4.0
6	8	4	4	5	3	5	2	3.8
7	13	4	4	4	4	4	4	4.0
8	4	5	5	4	4	n/a	4	4.4
9	6	4	4	5	4	4	3	4.0
10	9	7	6	6	7	1	7	5.7
Avg:		4.3	4.1	4.1	3.6	4.1	3.5	

TABLE I
PARTICIPANTS' RESPONSES TO EACH EXERCISE (E1 TO E6) ON A 7-POINT LIKERT SCALE. A LOW NUMBER INDICATES PREFERENCE FOR VERSION A; A HIGH NUMBER INDICATES A PREFERENCE FOR VERSION B.

The detailed results for each exercise are provided in Table I, while Figure 2 provides an overview of the aggregation of the results across all six exercises. This distribution was very surprising to us, as it was anticipated that version B would prove much easier to test. The distribution is roughly normal in shape, but with a strong tendency to the mean. From a quantitative perspective, we can make two observations:

- The distribution suggests that the participants judged version A and version B to be approximately equally difficult to test.
- Over half the responses received were 'neutral'. This suggests that the refactored version (version B) was not sufficiently distinct from version A, or that the differences involved did not affect testability.

A quantitative approach will not provide further insight, so we look more closely at the comments provided by the participants.

Exercise 1 and 2 were both based on the inheritance/delegation trade-off. Each participant gave these two

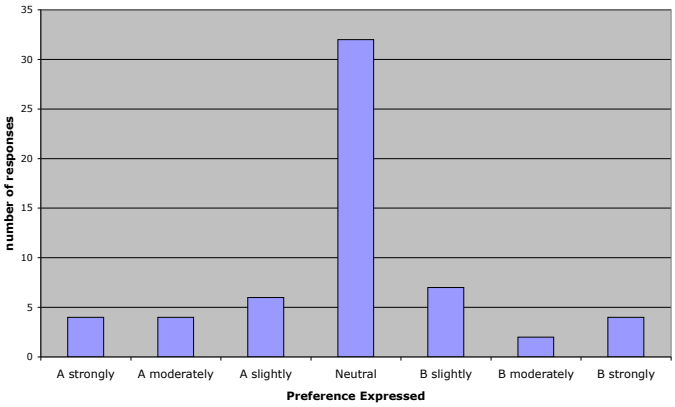

Fig. 2. Aggregation of participants' responses for all six exercises

exercises a similar rating (Spearman rank correlation 0.8), which validates our experimental approach to some degree. From the comments, most of the participants did not perceive a significant difference between testing functionality inherited from a superclass and functionality delegated to a client class. This flies in the face of the the conventional 'prefer delegation to inheritance' heuristic [12]. One explanation is that in the actual context of the simple Java application we built, there was little difference between the two designs. One participant observed that most IDEs display inherited attributes for a class, thus facilitating comprehension in version A.

Exercise 3 was based on testing a method that uses a field defined in a superclass versus one that uses a field defined in the same class. This was intended as a 'slam dunk' question in favour of version B, yet the participants' ranking varied considerably. One participant, who slightly favoured version B in this exercise, wrote a comment that summarised the authors' intentions thus: "In version B, you could see straight away how to set relevant fields. Although if you take into account Eclipse's prompts/shortcuts then both versions would be the same." However, most participants did not see this lack of cohesion as a problem. It is impossible that testing a method that accesses fields that are dispersed about the program is easier than testing a method that accesses only fields in the same class. Our suggested conclusion is that in this small example, the difference in testability between the two versions was not significant enough.

Exercise 4 was based on testing a method that has a parameter of a concrete class versus testing a method that uses a parameter of an appropriate interface type. This is an example of the 'program to an interface, not an implementation' edict [12]. One participant who strongly favoured version B wrote "Objects of Company class are much easier to test. My tests can pass anything that implements the InterIndustrialistPerson interface to the SetTheBoss setter method thus my tests can be more general." However, many participants did not distinguish between the versions, or even favoured version A. Although version B is a better design structure, the added level of indirection caused by the interface actually makes it a bit more work to test. Again, the benefits may be clear in a large system, but less evident in a small example.

Exercise 5 explored if it is easier to test a method in a simple class rather than one in a larger, more complicated class. Most of the participants did not regard the noise around the method under test to be an issue, though one did and commented "I felt a bit distracted by everything else going on in Student." One strongly preferred version A, as the method appeared lower in the class hierarchy, hence affecting fewer classes, a point overlooked by the authors.

Exercise 6 compared an inheritance relationship between two classes with two classes that share the same interface. The latter is the preferred solution, and from a testability perspective the effort required can be reduced by using parameterised test cases. Again, most participants did not see this as significant. One simply preferred the structure with concrete inheritance and one commented that a modern IDE hid the complexities of the relationships anyway. One participant who favoured version B noted that it might be easier to use mock objects in version B. A neutral participant commented "Having the interface is great for defining your test structure possibly quicker, but in reality I don't actually see a significant difference in implementation."

We experienced some challenges in designing and performing the experiment itself. To make it possible to complete the experiment in a reasonable time, we had to use a small application and direct the participants to certain aspects of it. This application transpired to be too simple in that many participants did not detect the differences between the original and refactored versions, although in an industrial-scale application these differences would surely be relevant. Although we asked participants to assess in each case which version was easier to test, many assessed them as equal because the resulting test case were identical, apparently ignoring the challenge in *writing* the test case, which was the key aspect of interest.

In terms of assessing our hypothesis that automated refactoring can be used to improve testability, we observe the following:

- Automated refactoring using the LSCC metric did lead to some significant design changes which could be anticipated to aid testability, viz,
  - Some inheritance relationships were changed to delegation.
  - The type of some attributes and parameters were changed from being a concrete class to an interface.
  - Methods that were separated from the fields they use (feature envy) were reunited.
  - Some classes that inherited from concrete superclasses were refactored sever the inheritance link and give both classes a shared interface instead.
- From a quantitative perspective, the survey results suggest that automated refactoring to improve testability is not effective. However, on looking at the written comments the participants made it is clear that many other factors were involved. As biased proponents of this approach, we take some comfort that the participant who spent

the longest time doing the exercises and gave the most detailed comments, found the refactored version B to be much easier to test.

A. Threats to Validity

This experiment was intended only to investigate whether automated improvement of testability is an area worthy of further study. As such, our results are not generalisable, for several reasons. Only one contrived Java application was used. The participants were self-selected, had an interest in unit testing and could not be considered a random sample of practicing software engineers as all are undertaking the same part-time programme of study.

The ordering of the test cases could tend to lead to a bias in favour of version B. After performing the exercise for version A, the participant would naturally find it easier to write a similar test case in version B. Judging from the participants' comments, this was not a significant factor, but varying the order of the versions, or randomly assigning each participant to only one version would have avoided this.

Selecting the parts of the program that were to be used as targets for the testability exercises could exert a bias on the results. Even if testability deteriorated overall, it might still be possible to find places where it improved. We were conscious of this potential for bias, but on inspecting the code we did not find parts that appeared to have become less testable in the refactored version.

VI. Future Work

Firstly, our approach to validation proved time consuming and not very effective. There are other approaches that can be used instead. Rather than asking experiment participants to judge the difficulty in writing test cases for the original and refactored versions of a program, we can use an automated test case generation tool to create the test cases. This can be applied to the original and refactored versions of the program and the resulting test suites compared on the basis of metrics such as code coverage, lines of test code and number of `assert` statements [4], [6]. The feasibility of this would need to be assessed in further research, but it has the advantage that it eschews the need for an costly experiment to assess the result.

Another approach is to focus on automated test case generation completely. In this case, the goal of the transformation is not to refactor the program for the benefit of the developers, but to transform it so as to enable more effective automated test case generation, while maintaining certain test adequacy criteria such as branch coverage or statement coverage[4]. Test cases then are generated for the transformed program, but applied to the original one. Harman discusses this idea and possible *testability transformations* in detail in [14]. It is interesting to consider if the application this type of this type of transformation can be driven using a search-based approach. The key questions are (1) is search necessary in this context? and (2) what would the fitness function look like? If testability can be improved by direct application of testability transformations, then search is not required. Defining an efficient fitness function is likely to be a challenge. One possibility is to generate test data for the program under transformation after each transformation, and to evaluate this. Performing search in this manner may be time consuming, though we note that a similar approach has been used e.g. in the configuration of application servers [24].

Naturally our approach can be extended to use other metrics rather than LSCC. From our experience with search-based refactoring, we would not anticipate that other cohesion metrics would produce a much better result that LSCC. However, combining LSCC with other cohesion metrics might result is some aspects of cohesion being covered that are not covered by LSCC alone.

We have not considered coupling metrics in this work but there is reason to expect that they would produce a useful result. For example, Brutnik and van Deursen [6] found a significant correlation between the Response For Class coupling measure [8] and test case size in the open source examples they studied. Probably a fitness function based on a judicious mixture of coupling and cohesion metrics would work well. They can be combined either using a weight-based approach [18] or a pareto-optimal approach [15].

VII. Conclusion

We have demonstrated through a proof-of-concept experiment that automated refactoring can improve the cohesion properties of a program in a way that would be expected to improve the program's testability. Our subsequent attempt to validate this through experimentation with industrial software engineers appeared to produce an ambivalent result. On closer analysis of the engineers' comments, our original hypothesis that automated refactoring can improve testability still appears likely to be valid, but requires further testing.

Acknowledgment

This research was partly funded by the Irish Programme for Research in Third-Level Institutions and by the Lero Graduate School in Software Engineering. The authors also thank the anonymous reviewers for their constructive feedback.

References

[1] Jehad Al Dallal and Lionel C. Briand. An object-oriented high-level design-based class cohesion metric. *Inf. Softw. Technol.*, 52:1346–1361, December 2010.

[2] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and Software Technology*, 51(9):1319 – 1326, 2009.

[3] Rushikesh Amin, Mel Ó Cinnéide, and Tony Veale. Laser: a lexical approach to analogy in software reuse. *Proceedings of the Workshop on Mining Software Repositories*, pages 112–116, 2004.

[4] A similar idea approach is used by Otero at al [21] to refactor a program as it is being evolved using genetic programming – the goal is not to improve program design in the usual sense, but to find a different design which may admit a useful transformation as part of the genetic programming algorithm.

[4] Linda Badri, Mourad Badri, and Fadel Toure. Exploring empirically the relationship between lack of cohesion and testability in object-oriented systems. In Tai-hoon Kim, Haeng-Kon Kim, Muhammad Khurram Khan, Akingbehin Kiumi, Wai-chi Fang, and Dominik lzak, editors, *Advances in Software Engineering*, volume 117 of *Communications in Computer and Information Science*, pages 78–92. Springer Berlin Heidelberg, 2010.

[5] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. *SIGSOFT Softw. Eng. Notes*, 20:259–262, August 1995.

[6] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *J. Syst. Softw.*, 79:1219–1232, September 2006.

[7] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*. IEEE, Sep 2010.

[8] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *Transactions in Software Engineering*, 20(6):476–493, June 1994.

[9] Steve Counsell, Stephen Swift, and Jason Crampton. The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Trans. Softw. Eng. Methodol.*, 15:123–149, April 2006.

[10] Jehad Al Dallal and Lionel C Briand. A precise method-method interaction-based cohesion metric for object-oriented classes. *IEEE TOSEM*, pages 1–37, Oct 2010.

[11] Ingo Feinerer and Kurt Hornik. *wordnet: WordNet Interface*, 2011. R package version 0.1-7.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Publishing, 1995.

[13] Tobias Gutzmann et al. Recoder: a framework for java program analysis and source code transformation, March 2010. http://sourceforge.net/projects/recoder.

[14] Mark Harman. Open problems in testability transformation. *IEEE International Conference on Software Testing Verification and Validation Workshop*, 0:196–209, 2008.

[15] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, pages 1106–1113, London, England, 7-11 July 2007. ACM.

[16] A.C. Jensen and B.H.C. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th annual conference on Genetic and Evolutionary Computation*, page 1341–1348. ACM, ACM, 07/2010 2010.

[17] Mark O'Keeffe and Mel Ó Cinnéide. Automated design improvement by example. In H. Fujita and D.M. Pisanelli, editors, *New Trends in Software Methodologies, Tools and Techniques*. The Netherlands: IOS Press, 2007.

[18] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, 2008.

[19] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring for software maintenance. *J. Syst. Softw.*, 81(4):502–516, 2008.

[20] Mark O'Keeffe and Mel Ó Cinnéide. A stochastic approach to automated design improvement. In James F Power and John T Waldron, editors, *International Conference on the Principles and Practice of Programming in Java*, pages 59–62, Kilkenny, June 2003.

[21] Fernando E. B. Otero, Colin G. Johnson, Alex A. Freitas, , and Simon J. Thompson. Refactoring in automatically generated programs. *Search Based Software Engineering, International Symposium on*, 0, 2010.

[22] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM.

[23] Harry M Sneed. Reengineering for testability. In *Proceedings of the 8th Workshop on Software Reengineering*. Gesellschaft fur Informatik, May 2006.

[24] Bowei Xi, Cathy H. Xia, Zhen Liu, Li Zhang, and Mukund Raghavachari. A smart hill-climbing algorithm for application server configuration. In *13th Int. Conference on the World-Wide Web*, pages 287–296, 2004.