Empir Software Eng
DOI 10.1007/s10664-016-9427-7

An experimental search-based approach to cohesion metric evaluation

Mel Ó Cinnéide[1] · Iman Hemati Moghadam[2] · Mark Harman[2] · Steve Counsell[3] · Laurence Tratt[4]

Abstract In spite of several decades of software metrics research and practice, there is little understanding of how software metrics relate to one another, nor is there any established methodology for comparing them. We propose a novel experimental technique, based on search-based refactoring, to 'animate' metrics and observe their behaviour in a practical setting. Our aim is to promote metrics to the level of active, opinionated objects that can be compared experimentally to uncover where they conflict, and to understand better the underlying cause of the conflict. Our experimental approaches include semi-random refactoring, refactoring for increased metric agreement/disagreement, refactoring to increase/decrease the gap between a pair of metrics, and targeted hypothesis testing. We apply our approach to

Communicated by: Massimiliano Di Penta

✉ Mel Ó Cinnéide
mel.ocinneide@ucd.ie

Iman Hemati Moghadam
i.moghadam@ucl.ac.uk

Mark Harman
mark.harman@ucl.ac.uk

Steve Counsell
Steve.Counsell@brunel.ac.uk

Laurence Tratt
laurie@tratt.net

[1] School of Computer Science, University College Dublin, Dublin, Ireland

[2] Department of Computer Science, University College London, London, England

[3] Department of Computer Science, Brunel University, London, UK

[4] Department of Informatics, King's College London, London, England

Published online: 13 April 2016

five popular cohesion metrics using ten real-world Java systems, involving 330,000 lines of code and the application of over 78,000 refactorings. Our results demonstrate that cohesion metrics disagree with each other in a remarkable 55 % of cases, that Low-level Similarity-based Class Cohesion (LSCC) is the best representative of the set of metrics we investigate while Sensitive Class Cohesion (SCOM) is the least representative, and we discover several hitherto unknown differences between the examined metrics. We also use our approach to investigate the impact of including inheritance in a cohesion metric definition and find that doing so dramatically changes the metric.

Keywords Refactoring · Software metrics · Empirical studies

1 Introduction

Like all engineers, software engineers want to measure the materials they engineer, leading to many proposals for ways to measure software using so-called 'software metrics' (Shepperd 1995). How do we know that our software metrics really do report numbers that reflect the true values of the property they claim to measure? This question may appear to be inherently unanswerable, but we believe it is possible to approach a partial answer from the angle of disciplined experimental metric cross validation. Previous work on metric validation has focussed on formal analysis (Weyuker 1988; Fenton 1994; Hitz and Montazeri 1996; Al Dallal 2010), empirical evaluation (Kemerer 1995; Counsell et al. 2005; Meyers and Binkley 2007; Beck and Diehl 2011) or user studies (Counsell et al. 2006; Bouwers et al. 2013; Simons et al. 2015). However, although these approaches can establish formal metric properties and assess their applicability and usability, they cannot determine if a metric actually measures the property it purports to measure.

In this paper we seek to approach this apparently unanswerable question through the concept of metric agreement and disagreement. That is, while we cannot ask whether a particular metric captures the true property it seeks to capture, we can ask whether it *agrees* with another metric that also claims to capture this property. Although this cannot tell us whether the metrics capture the elusive 'ground truth', it can increase or decrease our confidence in a disciplined manner. Our approach can be thought of as a form of cross validation that seeks to assess the degree to which metrics that ought to agree in theory, because they all purport to measure cohesion, really do agree in practice when applied to real-world software applications.

Many metrics purport to measure the same software property; our approach allows us to more rigorously investigate these claims. For example, many metrics have been introduced in the literature that aim to measure software cohesion (Lakhotia 1993; Bieman and Ott 1994; Harman et al. 1995; Fenton and Pfleeger 1996; Briand et al. 1996). If these metrics were measuring the same property, then they ought to produce, in some sense, similar results. This poses some important but uncomfortable questions: how do the results of metrics that purport to measure the same software property compare to one another? Can metrics that measure the same property disagree, and how strongly can they disagree? These questions are important, because we cannot rely on a suite of metrics to assess properties of software if we can neither determine the extent to which they agree, nor have any way to determine a likely worst case disagreement. They are also uncomfortable questions because, despite several decades of software metrics research and practice, there remains no answer, nor even an accepted approach to tackling them.

In this paper we address this problem by introducing a systematic experimental technique to assess software metrics. Our approach applies automated refactoring to a program, repeatedly measuring the values of a number of metrics before and after applying each refactoring. In this way we can make empirical observations about the relationships between the metrics. When a pair of metrics do not agree on the change brought about by a refactoring, we examine the causes of the conflict so as to gain a further (and more qualitative) insight into the differences between the metrics. Our focus is on metric conflict, because when two metrics agree they may or may not be measuring the same property, but when they are in conflict, we can conclude with confidence that they are not measuring the same property.

Note that we purposely do not attempt to define cohesion in this work. It is a commonly-used term in software development, and it is the cohesion metric definition themselves that attempt to provide a definition for the term. The fundamental goal in this work is to investigate these cohesion metric definitions and determine if they provide a consistent view of this software property.

We evaluate our approach on five widely-used metrics for cohesion, namely Tight Class Cohesion (TCC) (Bieman and Kang 1995), Lack of Cohesion between Methods (LCOM5) (Briand et al. 1998), Class Cohesion (CC) (Bonja and Kidanmariam 2006), Sensitive Class Cohesion (SCOM) (Fernández and Peña 2006) and Low-level Similarity-based Class Cohesion (LSCC) (Al-Dallal and Briand 2012). We perform these experiments using a search-based, metric-guided refactoring platform, Code-Imp (Hemati Moghadam and Ó Cinnéide 2011), that can apply a large number of refactorings without user intervention. Using Code-Imp, over 78,000 refactorings were applied to ten non-trivial, real-world Java programs comprising in total over 330,000 lines of code.

This paper is an extension of an earlier conference paper by the same authors (Ó Cinnéide et al. 2012). Its primary contributions are as follows:

1. We introduce a new approach to metric analysis at the source code level, implementing the approach to metric investigation using search-based refactoring first mooted by Harman and Clark (2004), but which has hitherto remained unimplemented. We introduce the concept of metric volatility, which measures the degree to which metrics are sensitive to syntactic program alterations achieved using automated refactoring. Using experimentally observed metric volatility, we develop an approach to systematic metric validation.
2. We demonstrate how our approach reveals that seemingly similar metrics can be in conflict with one another, and can pinpoint the source of the conflict thus providing new insight into the differences between the metrics.
3. We introduce a technique for iterative refactoring agreement and disagreement. Overall, we find that it is easier to force metric agreement than disagreement, which is expected since we study metrics that claim to measure the same program property: cohesion. Nevertheless, we also find that Sensitive Class Cohesion (SCOM) (Fernández and Peña 2006), defined in Fig. 3, is unusual because it is easier to force SCOM to disagree with the other metrics, providing evidence to suggest that this metric may be measuring something with which the others disagree.
4. We note that there may be a numeric gap in the values reported for two different cohesion metrics applied to the same system. We introduce techniques for gap opening and closing, using refactoring, which seek to widen (respectively reduce) the gap between metric values. This approach searches a metric's 'volatility space' to provide the decision maker with an informal 'confidence interval' for the 'true' metric value.

5. We present the results of a study on the impact of inheritance which yielded surprising results considering the current literature. Previous authors have argued that inheritance may optionally be considered in cohesion measurement, but attach little significance to the decision. Our experimental results refute this; including/excluding inheritance can have a large impact on the values reported by the metric.
6. We identify a number of undocumented anomalies in established cohesion metrics, thereby demonstrating the utility of our approach as a means of investigating metrics.

This paper extends our earlier conference paper (Ó Cinnéide et al. 2012) in several ways. We introduce two completely new techniques for comparing metrics in Section 5 and investigate them experimentally. Our earlier paper contained a preliminary investigation into the effect of including inheritance in a cohesion metric definition. In this journal paper this investigation is extended from considering four metric pairs to considering ninety metric pairs. Overall, the evidence base upon which our conclusions rest has been extended from approximately 3,000 refactorings to over 78,000 refactorings.

The remainder of this paper is structured as follows. In Section 2 we describe our experimental approach in more detail while in Section 3 we outline the platform we use in this paper to perform search-based refactoring. In Section 4 we describe our initial investigation into how a suite of software cohesion metrics changes in response to refactoring, which allows us to make general observations about the metrics. In Section 5 we perform focussed, pairwise metric comparisons involving refactoring to either iteratively increase/decrease metric agreement or to open/close the gap between metrics. In Section 6 we use our experimental approach to investigate a claim made in the metrics literature that including inheritance in a metric definition is optional and of little consequence. Section 7 describes related work while Section 8 discusses threats to the validity of our study. Finally, in Section 9, we summarise our conclusions and describe future work.

2 Motivation and Approach

The fundamental motivation for this work stems from a desire to 'animate' metrics and observe their behaviour in relation to each other in a practical setting. Rather than view them simply as passive measures to be extracted from source code, we aim to promote them to the level of active objects that have an opinion on what software quality is. We then want to play them off against each other, to ask their opinion on many software examples and to actively uncover where these metrics conflict with each other, and to explore these areas of conflict further to understand better why the metrics conflict.

In the coming subsections we motivate our choice of (1) refactorings to employ, (2) metrics to study and (3) software applications to employ in our experiments. Finally we provide a high-level overall summary of the experiments that are detailed later in the paper.

2.1 Motivation for the Choice of Refactorings

A single software application allows only one set of metric measurements to be made. This is clearly not enough to make meaningful comparisons. A Source Code Management system such as Subversion or Git provides multiple commits of a software application and so serves as a better basis for comparison, and many studies have taken this approach (Succi et al. 2005; Counsell et al. 2006; Al-Dallal and Briand 2012). However, a sequence of commits of a software application may vary dramatically in terms of how great the

gap is between each commit. This lack of control over the differences between the commits is a significant confounding factor in studies that use software repositories to compare software metrics.

Our approach to this problem begins with the observation that individual refactorings in the style of Fowler et al. (1999) involve small behaviour-preserving program changes that typically have an impact on the values of software metrics that would be calculated for the program. However, whether a refactoring will improve or disimprove a metric is not, in general, knowable *a priori*. For example, in applying the Push Down Method refactoring, a method is moved from a superclass to those subclasses that require it. The superclass may become more cohesive if the method moved was weakly connected with the rest of the class. It may instead become less cohesive, if the moved method served to glue other methods and fields of the superclass together. It is impossible to state that the Push Down Method refactoring leads to an improvement or a deterioration in cohesion without examining the context in which it is being applied. Furthermore, the impact the refactoring will have on the metric will depend on the precise notion of cohesion that the metric embodies.

There are two other reasons why we chose to use refactorings. Firstly, using refactorings means that all versions analysed are the same program in terms of input/output mapping and hence have the same domain complexity. The difference in complexity between the versions is thus completely due to the complexity of the software implementation. It is of course trivial and uninteresting to improve software metric values by removing functionality from the program. By using refactorings, we ensure that this does not happen. Secondly, using refactorings gives us control over what is measured. Rather than measure commits to a source code repository, we can apply refactorings that actively take us to areas of metric conflict and shun less interesting refactorings upon which all metrics agree.

Turning now to the set of refactorings we employ, these are described in complete detail in Section 3.3. They represent a broad set of refactorings that can move methods and fields around a class hierarchy as well as create new classes and merge existing ones. The 'change visibility' refactorings do not change cohesion directly, but may enable other refactorings that do change cohesion, e.g. increasing the visibility of a field may permit a method that uses this field to be pushed down to a subclass. We choose not to use refactorings that split methods, as such refactorings have many options and the search space is rich enough already. We omitted the general MoveMethod refactoring as its preconditions are quite strict and involve much source code detail. It is worth noting in this context that bugs exist in commercial implementations of even 'simple' refactorings like Rename (Schäfer et al. 2012; Gligoric et al. 2013), and as we need to apply thousands of refactorings without programmer intervention, it was important that our refactorings work correctly.

2.2 Motivation for the Choice of Metrics

The methodology we use in this paper can be applied to any metrics set. We decided to study cohesion metrics as they are generally regarded as more challenging than coupling metrics and a generally accepted informal definition of cohesion continues to elude the object-oriented software engineering community (Counsell et al. 2006). There remains many cohesion metrics to choose from. Our primary guiding principle in choosing cohesion metrics to study was to select a range of metrics that reflects the broad field of cohesion metrics research. Accordingly the metrics we study are as follows:

1. Tight Class Cohesion (TCC) (Bieman and Kang 1995) is the 'elder' of the metrics we study. It is one of the first object-oriented cohesion metrics proposed and can be taken as a traditional view of cohesion.
2. Low-level Similarity-based Class Cohesion (LSCC) (Al-Dallal and Briand 2012) is by contrast one of the newest cohesion metrics that was available at the time of writing.
3. Lack of Cohesion between Methods (LCOM) is a member of the best-known object-oriented metrics suites, that of Chidamber and Kemerer (1994). This metric was the object of much criticism (Kitchenham 2010), so we chose to use a more refined form of this metric, namely LCOM5 (Briand et al. 1998).
4. Sensitive Class Cohesion (SCOM) (Fernández and Peña 2006) is a cohesion metric that the authors claimed to be more sensitive than those previously reported in the literature. This made it an interesting choice to use in our comparisons.
5. Class Cohesion (CC) (Bonja and Kidanmariam 2006) is a well-cited cohesion metric that was developed around the same time as SCOM and is similarly claimed to improve on previous cohesion metrics.

The formulae for these metrics are presented in Section 3.4. It is worth nothing that for TCC, LSCC, SCOM and CC, an increase in value means an improvement in cohesion. However LCOM5 measures *lack* of cohesion and so an increase in value means a disimprovement in cohesion. While we could have used $(1 - LCOM5)$ in place of LCOM5 to circumvent this anomaly, this could also have led to confusion. Where necessary in the subsequent analysis, we restate this fact about LCOM5.

2.3 Motivation for the Choice of Software Applications

Table 1 provides an overview of the software applications that are used in the experiments in this paper. They are all open source systems, as closed source systems are harder to obtain and subject to licensing limitations. Our main aim in choosing these applications is to select as broad a range as possible. Thus the application domains of the chosen applications are very diverse and the application sizes range from 12K to 87K. JHotDraw was purposely chosen as it often features in this type of study, but the remaining choices were semi-random. In particular, we did not apply any pre-test to determine how cohesive the applications were initially or how many refactorings could be applied to them.

Table 1 Software applications studied in this paper

System	Description	LOC	#Classes
JHotDraw 5.3	Graphical GUI framework	14,577	208
XOM 1.1	XML object model API	28,723	212
ArtofIllusion 2.8.1	3D modelling	87,352	459
GanttProject 2.0.9	Project scheduling	43,913	547
JabRef 2.4.2	Bibliography manager	61,966	675
JRDF 0.4.1.1	API for RDF	12,773	206
JTar 1.2	Compression library	9,010	59
mxGraph 1.5.0.2	Interactive graphing	48,810	229
HTMLUnit 1.4	Unit testing	12,297	194
JSMPP 2.1.0	SMPP protocol	10,923	144

2.4 Overview of Experiments

The general approach taken in this paper is to measure a set of metric values on a program, and then apply a sequence of refactorings to the program, measuring the metrics again after each refactoring is applied. Each refactoring represents a small, controlled change to the software, so it is possible to identify patterns in how the metric values change, and how they change in relation to each other. For N refactorings and M metrics, this approach yields a matrix of $(N + 1) \times M$ metric values, which we term the *refactoring/metric matrix*. As will be demonstrated in Sections 4, 5 and 6, this matrix can be used to make a comparative, empirical assessment of the metrics and to detect areas of metric disagreement that can be subjected to closer examination. In the coming paragraphs we provide a very high-level overview of the three investigations we perform in this work.

In Investigation I in Section 4 we throw all the metrics in together and take the subject software application on a long 'refactoring walk', observing how the metrics change after every applied refactoring. We do this to obtain an overview of how volatile the metrics are under refactoring and to assess how much the metrics conflict. The search is not strongly-directed: if a proposed refactoring improves at least one of the metrics being studied, it is accepted. Such refactorings prove easy to find, so we obtain long refactoring sequences that suit our purposes well.

In Investigation II in Section 5 we turn our focus to *pairwise* comparison of metrics. Here we take a pair of metrics and play them against each other to actively determine to what extent they agree and disagree. The search for refactorings here is, in contrast to Investigation I, very strongly directed. The four experiments we perform are:

1. Iterative Refactoring Agreement (IRA). Find as long a refactoring sequence as possible where both metrics agree that each refactoring improves cohesion.
2. Iterative Refactoring Disagreement (IRD). Find as long a refactoring sequence as possible where, on each refactoring, both metrics disagree on whether or not it improves cohesion.
3. Gap Opening (GOR). Find as long a refactoring sequence as possible where the gap between the two metric values increases with each refactoring.
4. Gap Closing (GCR). Find as long a refactoring sequence as possible where the gap between the two metric values decreases with each refactoring.

IRA and IRD provide a more incisive view of the similarity between the two metrics. Similar metrics will exhibit high values of IRA and low values of IRD; dissimilar metrics the opposite. The combined values of IRA and IRD for a metric pair provide a sense of the extent to which the two metrics embody a similar notion of cohesion. On the other hand, GOR and GCR say more about how 'bendy' the two metrics are for a particular application, i.e. to what extent they can be forced by refactoring to agree or disagree. This provides an insight into how fundamental the difference between the two metrics is for the application being investigated.

In Investigation III in Section 6, we illustrate another use case for our approach to metrics assessment, i.e. to investigate unverified hypotheses in the metrics literature that hitherto could not be experimentally tested. The example we use is the the issue of whether or not inherited attributes should be considered in a metric definition. Thus we create 'inherited' versions of the employed metrics and experimentally compare them with their 'normal' versions. The search for refactorings here is, again, very strongly directed. We seek refactorings that improve one metric, and then measure the effect on the other metric. This enables us

to conclude that including inherited attributes in a metric definition has a significant impact on a cohesion metric. More importantly, it demonstrates that this methodology can be used to investigate other such hypotheses about metrics.

3 The Code-Imp Platform

Code-Imp is an extensible platform for metrics-driven search-based refactoring that has been previously used for automated design improvement (O'Keeffe and Ó Cinnéide 2008a, b; Hemati Moghadam and Ó Cinnéide 2011, 2015). It provides design-level refactorings such as moving methods around the class hierarchy, splitting classes and changing inheritance and delegation relationships, but does not support low-level refactorings that split or merge methods. Code-Imp was developed on the RECODER platform (Gutzmann et al. 2013). In the following subsections we describe further the architecture of Code-Imp (Section 3.1), the search algorithm employed on this work (Section 3.2), the refactorings employed (Section 3.3), and finally the software metrics used in the creation of the fitness function that guides the search (Section 3.4).

3.1 Code-Imp Architecture

The overall architecture of the Code-Imp framework as it is used in this work is depicted in Fig. 1. Code-Imp first parses the program to be refactored to produce a set of Abstract Syntax Trees (ASTs). It then repeatedly chooses a refactoring and attempts to apply this refactoring to the ASTs. If the precondition of the refactoring fails, then the refactoring is not applied. If all is well and the refactoring is applied successfully, then the new values for the cohesion metrics under investigation are measured. If these metric values comply with the type of investigation that is being performed, then they are recorded; if not, then

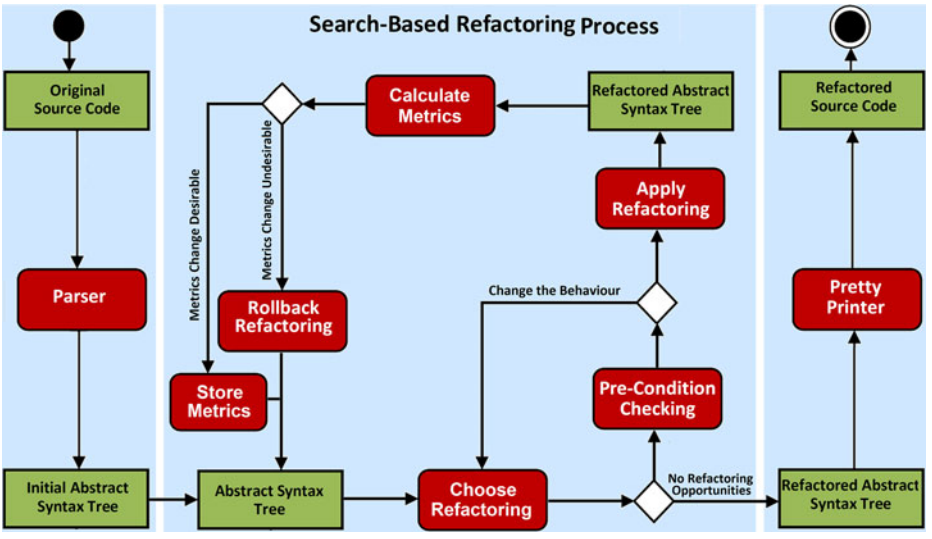

Fig. 1 Architecture of the Code-Imp search-based refactoring framework

the refactoring is rolled back. Code-Imp then chooses another refactoring to apply and the process is repeated.

The process whereby a refactoring is chosen is a variant on hill climbing and is described in Section 3.2. Determining if the metric values comply with the type of investigation that is being performed is defined in the context of each different investigation; see Sections 4, 5 and 6. When the process completes, either because a long enough refactoring sequence has been generated or because no more applicable refactorings can be found, Code-Imp pretty-prints the source code and the entire refactoring process stops.

3.2 Search Algorithm

The search-based algorithm we employ to perform the refactoring is defined in Fig. 2. It is stochastic, as the `pick` operation makes a random choice of the class to be refactored, the refactoring type to be used and the actual refactoring to be applied. It is only necessary to run this search once on each software application, as each refactoring applied is a complete experiment in itself. The purpose of this algorithm is to give each class an equal chance of

Input: set of classes in program being refactored.
Input: set of 14 refactoring types (e.g. Pull Up Method).
Input: set of metrics to be analysed.
Output: refactoring/metric matrix.

```
refactoring_count = 0
repeat
    classes = set of classes in program
    while !empty(classes) do
        class = classes.pick()
        refactoring_types = set of refactoring types
        while !empty(refactoring_types) do
            refactoring_type =refactoring_types.pick()
            refactorings.populate(refactoring_type, class)
            while !empty(refactorings) do
                refactoring = refactorings.pick()
                refactoring.apply()
                if fitness_function_improves() then
                    refactoring_count++
                    update refactoring/metric matrix
                    break
                else
                    refactoring.undo()
                end
            end
        end
    end
until (refactoring_count == desired_refactoring_count);
```

The functions used in this algorithm are defined as follows:
Set<element>::pick: removes and returns a *random* element from a set;
Set<Refactoring>::populate(type, class): adds to the set all legal refactorings of the given type that can be applied on the given class;
fitness_function_improves: Tests if the applied refactoring has improved the software metrics. Details vary between the investigations, as described in Sections 4, 5 and 6.

Fig. 2 The search-based refactoring algorithm used to explore software metrics

being refactored and to give each refactoring type (Pull Up Method, Collapse Hierarchy, etc.) an equal chance of being applied. This is important in order to reduce the risk that bias in the refactoring process affects the observed behaviour of the metrics. The output of this algorithm is the refactoring/metric matrix as described in Section 2.4. The details of the fitness function are not defined in this algorithm, as they depend on the exact nature of what is being investigated. The fitness functions will be defined in Sections 4, 5 and 6 where the experiments are described in more detail.

3.3 Refactorings Employed

Code-Imp currently implements the following refactorings (Fowler et al. 1999):

Method-level Refactorings

Push Down Method	Moves a method from a class to those subclasses that require it.
Pull Up Method	Moves a method from a class(es) to its immediate superclass.
Decrease/Increase Method Accessibility	Changes the accessibility of a method by one level, e.g. public to protected or private to package.

Field-level Refactorings

Push Down Field	Moves a field from a class to those subclasses that require it.
Pull Up Field	Moves a field from a class(es) to their immediate superclass.
Decrease/Increase Field Accessibility	Changes the accessibility of a field by one level, e.g. public to protected or private to package.

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.

3.4 Software Metrics and Fitness Function

In Section 3.2 above we describe how our search process tries to create a random refactoring sequence. However, randomly-chosen refactorings are likely to cause all software metrics to deteriorate, which is not of interest. In order to address this, we use the software metrics that are being studied to guide and control the refactoring process itself. The precise formulation of the fitness function depends on the investigation taking place. For example, in Investigation I we apply a refactoring only if it improves *at least one* of the metrics being studied. This ensures that each accepted refactoring will improve the cohesion of the program in terms of at least one of the metrics, though it may, in the extreme case, worsen it for all the other metrics.

The fitness function that guides the search thus is a computation based on one or more software metrics. Code-Imp provides two implementations for each metric related to the inclusion or exclusion of inheritance in the definition of the metric (we return to this issue in Section 6). Five cohesion metrics are used in this paper, namely Tight Class Cohesion (TCC) (Bieman and Kang 1995), Lack of Cohesion between Methods (LCOM5) (Briand et al. 1998), Class Cohesion (CC) (Bonja and Kidanmariam 2006), Sensitive Class Cohesion (SCOM) (Fernández and Peña 2006) and Low-level Similarity-based Class Cohesion (LSCC) (Al-Dallal and Briand 2012). The definitions of these metrics are presented in Fig. 3.

As with all automated approaches, the refactoring sequence generated by Code-Imp may not resemble the refactorings that a programmer would be inclined to undertake in practice. This issue is not relevant here as our focus is on the *changes in the metric values*, rather than the design changes brought about by the refactorings themselves.

4 Investigation I: General Assessment of Cohesion Metrics

In this investigation we take a 'refactoring walk' through the landscape of the range of cohesion metrics under consideration. Our goal is to gain an overall understanding of how the metrics change, and to seek out possible anomalous behaviour that can be investigated further.

As explained in Section 2, random application of refactorings will usually cause deterioration in all cohesion metrics. We therefore use a search that cycles through the classes of the program under investigation as described in Fig. 2, and tries to find a refactoring on the class that improves *at least one* of the metrics being studied. The search will apply the first refactoring it finds that improves any metric. The other metrics may improve, stay the same, or deteriorate. Because this fitness function is easy to improve, we obtain the long refactoring sequences that are required to draw conclusions about relationships between metrics.

The metrics formulae presented in Fig. 3 show how to calculate the metric value for a single class. How to measure the cohesion of an entire program is not so clear. One can simply take the average cohesion of all the classes in the program, but this can create anomalies because each class carries the same weight regardless of its size. Al-Dallal and Briand (2012) address this issue by weighting each class according to its size, using a size measure that corresponds to the metric. For the LSCC metric, they use a weight of $l * k * (k - 1)$ where l is the number of attributes in the class, and k is the number of methods in the class. As the LSCC metric measures the number of shared attributes between each pair of metrics, this weighting is appropriate. This weighting also makes sense for SCOM and CC, as these

$\text{LSCC}(c) = \begin{cases} 0 & \text{if } l{=}0 \text{ and } k > 1, \\ 1 & \text{if } (l > 0 \text{ and } k{=}0) \text{ or } k{=}1, \\ \sum_{i=1}^{l} x_i(x_i-1)/lk(k-1) & otherwise. \end{cases}$	The cohesion between two methods is the collection of their direct and indirect shared attributes.
$\text{TCC}(c) = \frac{\lvert\{m1, m2 \in M_I(c) \wedge m1 \neq m2 \wedge \text{cau}(m1, m2)\}\rvert}{k(k-1)/2}$	Two methods are cohesive if they both use a common attribute of the class c, directly or indirectly.
$\text{CC}(c) = 2\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} \frac{\lvert I_i \cap I_j\rvert}{\lvert I_i \cup I_j\rvert}/k(k-1)$	The cohesion between two methods is the ratio of the collection of their shared attributes to the total number of their referenced attributes.
$\text{SCOM}(c) = 2\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} \frac{\lvert I_i \cap I_j\rvert}{\min(\lvert I_i\rvert, \lvert I_j\rvert)} * \frac{\lvert I_i \cup I_j\rvert}{l}/k(k-1)$	The cohesion between two methods is the ratio of the number of attributes they share to the minimum number of their referenced attributes, weighted by the total number of attributes referenced.
$\text{LCOM5}(c) = \frac{k - \frac{1}{l}\sum_{a \in A_I(c)} \lvert\{m \mid m \in M_I(c) \wedge a \in \text{I}_\text{m}\}\rvert}{k-1}$	Measures *lack* of cohesion of a class in terms of the proportion of attributes each method references. Unlike the other metrics, a lower value indicates better cohesion.

In the above: c is a particular class; $M_I(c)$ is the set of methods implemented in c; $A_I(c)$ is the set of attributes implemented in c; k and l are the number of methods and attributes implemented in class c respectively; I_i is the set of attributes referenced by method i; x_i is the number of 1s in the ith column of the Method-Attribute Reference (MAR) matrix, $\text{MAR}(i, j)$ holds 1 if ith method directly or indirectly references jth attribute; $\text{cau}(m1, m2)$ holds 1 if $m1$ and $m2$ use an attribute of class c in common.

Fig. 3 Formal and informal definitions of the cohesion metrics evaluated in this paper

metrics are also based on attribute access by methods. TCC does not care about the number of attributes two methods access, so we use a simpler weighting of $k * (k-1)$. The LCOM5 metric is concerned with the number of attributes each method accesses (shared accesses are irrelevant), so in this case $k * l$ is a suitable weight.

Most software metrics are ordinal in nature, so computing the difference between two metric values or the the average of several metric values are theoretically dubious operations. However, our experience suggests that these metrics are not far from being on an interval scale and so the risk in treating them as interval scale metrics is slight in relation to the advantages that accrue. Briand et al. make a similar argument for the use of parametric methods for ordinal scale data (Briand et al. 1996).

4.1 Results and Analysis

We applied this refactoring process to the first eight open source Java projects presented earlier in Table 1. In each case, the experiment was allowed to run for five days, or until a sequence of over 1000 refactorings was reached. In total, 3,451 refactorings were applied, as shown in Table 2 (note that in later experiments in Sections 5 and 6, we use more applications and larger numbers of refactorings).

The applications were of high quality initially, so improvements to cohesion were time-consuming to find. To elaborate this point, in the space of all possible implementations of a program, we can expect any human-implemented solution to be of relatively high quality. For example, a monolithic system implemented as class with a thousand methods would no doubt permit many refactorings that improve the employed cohesion metrics, but such enormous and poorly-designed classes do not occur in practice.

JHotDraw proved the easiest program to refactor because it has a rich inheritance hierarchy that provided plenty of opportunities to refactor. Note that in this work we are using the refactoring process only to investigate the properties of the metrics and the potential for those metrics to measure program quality. We make no claim that the refactored program has a better design than the original program.

4.1.1 Volatility

One aspect of a metric that this investigation allows us to see is its *volatility*. A volatile metric is one that is changed frequently by refactorings, whereas an *inert* metric is one that is changed infrequently by refactorings. Thus metric M_1 is more volatile than metric M_2 if, over the course of a refactoring sequence, M_1 is more frequently changed by a refactoring than M_2. Our approach exploits this observation to develop experimental techniques to investigate the agreement and disagreement between software metrics as a means of systematically cross-validating metrics. Our approach provides a way to gather experimental evidence that enables observations such as these:

1. 'These two metrics do not appear to be measuring the same thing';
2. 'Metric M_1 appears to be measuring something additional to that measured by metrics M_2, M_3, M_4 and M_5';
3. 'The apparent disagreement between these two metrics for program P appears to be largely superficial';

Table 2 Metric volatility as a percentage

	JHotDraw (1005)	JTar (115)	XOM (193)	JRDF (13)	JabRef (257)	JGraph (525)	ArtOfIllusion (593)	Gantt (750)	All (3451)
LSCC	96	99	100	92	99	100	99	96	98
TCC	86	53	97	46	61	72	84	71	78
SCOM	79	70	93	92	79	89	77	80	81
CC	100	98	100	92	99	100	100	99	100
LCOM5	100	100	100	100	100	100	100	99	100

This shows the percentage of refactorings that caused a change in a metric. The number in parentheses is the total number of refactorings that were performed on this application.

4. 'Metric M_1 is least sensitive to superficial syntactic changes in the program being measured';
5. 'Of all the metrics in a metrics suite, M_1 is the metric that appears to be most consistent with the others'.

Volatility is an important factor in determining the usefulness of a metric. For example, in search-based refactoring, a highly volatile metric will have a very strong impact on how the refactoring proceeds while a relatively inert metric may simply be pointless to compute. In a software quality context, measuring the improvement in a system's design using a set of inert metrics is likely to be futile, as they are, by definition, crude measures that do not detect subtle changes in the property they measure. Furthermore, an inert software metric does not, by definition, have the desirable property of *actionability* (Fenton 1994), so that when it yields a poor value, it cannot easily be improved by refactoring. Table 2 shows the volatility of the five metrics in each individual system under investigation, and averaged across all systems. Note that we only present the *direction* of change in a metric and not the *amount* of change, as calculating the arithmetic difference between ordinal scale metrics is not a meaningful operation.

The first observation is that LSCC, CC and LCOM5 are all highly volatile metrics. In 99 % of the refactorings applied across all applications, each these metrics either increased or decreased. The relative lack of volatility of the TCC metric is largely due to the `cau` relation (see Fig. 3), which holds relatively rarely for any given pair of methods.

The results for the JRDF application are notable. All metrics except TCC are highly volatile for this application. Although JRDF is one of the larger applications, a total of only 13 refactorings could be applied to it, compared to the 1000+ refactorings that could be applied to JHotDraw, a similarly-sized application. The explanation for this lies in the nature of the applications. In JHotDraw, 86 % of the classes are subclasses, whereas in JRDF this figure is only 6 %. Since most of the refactorings Code-Imp applies relate to inheritance, an application that makes little use of inheritance provides few opportunities to refactor.

While there is some consistency across the different applications, the JRDF example illustrates that, given an individual metric, volatility can vary substantially between systems. We attempted normalising the volatilities against the overall volatility of each application, and, while this improved the consistency somewhat, a large variance remained. We thus conclude that volatility is dependent on a combination of a metric and the application to which it is applied. To what extent the nature of the application or metric determines volatility is beyond the scope of this paper, but is an interesting avenue for further exploration.

Table 3 Of those refactorings that change a metric, the percentage that are improvements and disimprovements, i.e., an up-arrow indicates an improvement in cohesion, a down-arrow indicates a disimprovement in cohesion

	JHotDraw	JTar	XOM	JRDF	JabRef	JGraph	ArtIllusn	Gantt	Avg
LSCC	↑50 , 46↓	↑50 , 49↓	↑57 , 43↓	↑46 , 46↓	↑54 , 46↓	↑51 , 48↓	↑57 , 42↓	↑53 , 43↓	↑53 , 45↓
TCC	↑45 , 41↓	↑30 , 23↓	↑51 , 46↓	↑23 , 23↓	↑34 , 27↓	↑37 , 35↓	↑52 , 35↓	↑39 , 31↓	↑43 , 35↓
SCOM	↑38 , 40↓	↑34 , 36↓	↑50 , 44↓	↑46 , 46↓	↑37 , 42↓	↑36 , 53↓	↑44 , 33↓	↑40 , 40↓	↑40 , 41↓
CC	↑53 , 47↓	↑52 , 46↓	↑51 , 49↓	↑46 , 46↓	↑54 , 44↓	↑61 , 39↓	↑58 , 42↓	↑57 , 42↓	↑56 , 44↓
LCOM5	↑51 , 49↓	↑50 , 50↓	↑48 , 52↓	↑54 , 46↓	↑49 , 50↓	↑41 , 59↓	↑56 , 43↓	↑50 , 50↓	↑50 , 50↓

4.1.2 Probability of Positive Change

Table 2 shows how volatile the metrics are, but it does not show whether the volatility is in a positive or negative sense. In Table 3 we present this view of the metrics. Recall that every refactoring applied in this investigation improves at least one of the cohesion metrics. It is remarkable then to note how often an improvement in one cohesion metric leads to a deterioration in another. Taking LSCC and ArtOfIllusion as an example, LSCC decreases in 42 % of the refactorings (593 in total). So for ArtOfIllusion, 249 refactorings that improved at least one of TCC, SCOM, CC or LCOM5, as guaranteed by the refactoring process, caused LSCC to worsen.

This pattern of conflict is repeated across Table 3. As summarised in Table 4, TCC, LSCC and SCOM exhibit collective moderate positive correlation, while CC and LCOM5 show mixed correlation ranging from moderate positive correlation (LCOM5 and SCOM) to strong negative correlation (LCOM5 and CC).

In order to summarise the level of disagreement across the set of metrics, we also considered each pairwise comparison between each pair of metrics for each refactoring. For five metrics we have $(5 * 4)/2 = 10$ pairwise comparisons per refactoring. For 3,451 refactorings, this yields a total of 34,510 pairwise comparisons. Each pair is categorised as follows:

Agreement: Both metric values improve, both deteriorate, or both stay the same.
Dissonant: One value improves or deteriorates, while the other stays the same.
Conflicted: One value improves, while the other deteriorates.

Across the entire set of refactorings, we found the levels to be as follows: 45 % agreement, 17 % dissonant and 38 % conflicted. The figure of 38 % conflicted is remarkable and indicates that, in a significant number of cases, what one cohesion metric regards as an improvement in cohesion, another cohesion metric regards as a decrease in cohesion. This has a practical impact on how cohesion metrics are used. Trying to improve a software system using a combination of conflicted cohesion metrics is impossible — an improvement in terms of one cohesion metric is likely to cause a deterioration in terms of another metric.

4.2 Summary

This investigation has served to show the variance between software cohesion metrics in terms of their volatility and their propensity to agree or disagree with each other. Of course a cohesion metric that completely agrees with another makes no contribution to the

Table 4 Spearman rank correlation between the metrics across all refactorings and all applications

	LSCC	TCC	SCOM	CC
TCC	0.60			
SCOM	0.70	0.58		
CC	0.10	0.01	−0.28	
LCOM5	−0.17	−0.21	−0.46	0.72

Note that LCOM5 measures *lack* of cohesion, so a negative value indicates positive correlation.

cohesion debate. However, the conflict between the metrics indicates that the suite of cohesion metrics do not simply reflect *different* aspects of cohesion, they reflect *contradictory* interpretations of cohesion.

Note that the conflict observed is also a function of the types of refactorings employed. A different pattern of conflict/agreement would be expected if other refactoring types were added to the refactorings used in the experiments. However, the refactoring types we use are standard ones, and, while the addition of other refactoring types would provide a more accurate picture of the extent of the metric disagreement, the metric conflict engendered by the refactoring types we employ would remain.

In order to investigate this conflict further, we conduct detailed analyses on pairs of metrics to further explore how they conflict during search-based refactoring. The results of this are presented in the following section.

5 Investigation II: Pairwise Comparison of Cohesion Metrics

In this section we use our experimental approach to compare and contrast *pairs* of metrics in a focussed way. The goal is to find areas of anomaly that will enable us to gain insight into the metrics themselves. We investigated each of the five metrics presented in Fig. 3, comparing each one against each of the other metrics. The 10 software applications we analysed are as presented in Table 1.

The two areas we explored were *Iterative Refactoring Agreement/Disagreement* (IRA and IRD) and *Gap Opening/Closing Refactoring* (GOR and GCR). To define these precisely, let us assume that we are comparing metrics A and B. When a refactoring is applied to a program P it may either increase ($\uparrow$), decrease ($\downarrow$) or not affect ($=$) each of A and B for the program P. A **metric profile** is a set of pairs that define when a refactoring will be accepted during an experiment. For example $\{(A \uparrow, B =), (A \downarrow, B \downarrow)\}$ denotes that we accept refactorings that increase A and do not affect B, and refactorings that cause both A and B to decrease.

The metric profiles for the experiments in this section are presented in Table 5. Intuitively, IRA seeks refactorings that cause both metrics to increase. We elected not to include in IRA cases where neither metric changes, as we anticipated that many such metric pairs would be found, thus obscuring the more interesting cases where both metrics improve. IRD seeks to increase metric A while B decreases or remains the same. GOR seeks to increase the arithmetic gap between metric A and B, while GCR seeks to reduce this gap. This metric profile is transformed into a fitness function and used in the search algorithm

Table 5 Metric profiles

	Metric Profile
IRA	$\{(A \uparrow, B \uparrow)\}$
IRD	$\{(A \uparrow, B \downarrow), (A \uparrow, B =)\}$
GOR	$\{(A =, B \downarrow), (A \uparrow, B \downarrow), (A \uparrow, B =), (A \uparrow, B \uparrow)^*, (A \downarrow, B \downarrow)^*\}$
GCR	$\{(A =, B \uparrow), (A \downarrow, B \uparrow), (A \downarrow, B =), (A \downarrow, B \downarrow)^*, (A \uparrow, B \uparrow)^*\}$

For GOR and GCR we assume that A is greater than B prior to refactoring application. *A refactoring is accepted only if it increases the gap between A and B (for GOR) or reduces the gap (for GCR).

presented in Fig. 2. We provide experimental results and analysis in the subsequent subsections.

5.1 Iterative Refactoring Agreement and Disagreement (IRA and IRD)

We compared each pair of metrics from the set of five metrics defined in Fig. 3; 10 comparisons were therefore required. Each experiment was performed on the 10 software applications described in Table 1. To investigate IRA therefore required 100 experiments. Note that IRA is defined as a refactoring sequence where both metrics improve for each refactoring, and therefore it must thus be performed separately for each refactoring pair. Also, since LCOM5 measures *lack* of cohesion, IRA accepts refactorings that decrease LCOM5 and increase the other metric; IRD is similarly amended to take this idiosyncrasy of LCOM5 into account.

In the case of IRD, each metric pair (A, B) has to be analysed *twice*, once using $\{(A\uparrow, B\downarrow), (A\uparrow, B=)\}$ to lead the search and then using $\{(B\uparrow, A\downarrow), (B\uparrow, A=)\}$ to lead the search. If we attempted to merge these into a single experiment, the refactoring process would be inclined to perform and undo the same refactoring multiple times. In total therefore the data presented in this subsection is derived from 300 experiments, involving the application of a total of 19,473 refactorings.

Figure 4 presents an overall view of the results of the IRA and IRD experiments. A number of general observations can be made from this data. For every metric, increasing its agreement with the other metrics overall proved easier than increasing disagreement overall. Nevertheless, a considerable amount of disagreement appears in every case. LSCC is the metric that other metrics most agreed with, as seen in the LSCC '+' column, and is the metric that most agreed with other metrics, as evidenced by the shaded component at the bottom of the '+' column for each of the other metrics. In some sense then, LSCC is the best single representative of this set of metrics.

We now turn our attention to the most inert metric, SCOM. For every metric, distinctly more refactorings were found to increase disagreement with SCOM than were found to increase agreement (as evidenced by the green bars in the '+' and '−' columns and the bars in the SCOM '−' column), and SCOM stands alone in this regard. So it is easy to make SCOM disagree with the other metrics, and hard to make it agree with them. We explored

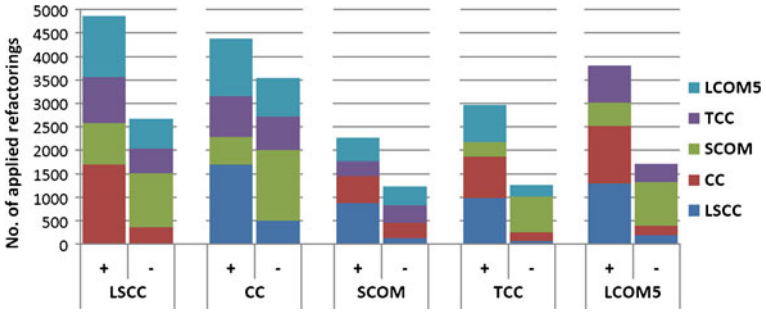

Fig. 4 An overall view of the number of refactorings applied in measuring IRA and IRD. Agreement is depicted by '+' while disagreement is depicted with a '−'. The '−' column for a metric depicts the disagreement that occurred while that metric was being improved. Agreement between each two metrics is measured once and the result appears in the '+' column for both metrics

this disagreement further and found that in 81 % of cases where SCOM improved, the other metric actually disimproved rather than simply remaining the same (19 % of cases). The pattern of low agreement and high conflict between SCOM and the other metrics indicates that SCOM detects aspects of cohesion that the other metrics largely disagree with, and vice versa. In a sense then, SCOM is the least representative of this set of metrics.

Looking in more detail at the refactorings that involve SCOM, the following observations can be made:

- SCOM prefers fields to reside in the class where they are used, so whenever another metric approved a refactoring that e.g. pulled up a field to a superclass, SCOM would deteriorate as the field is now separated from the methods that access it.
- CC prefers fields to be in superclasses and accessed from subclasses, thus conflicting directly with SCOM in many cases.
- TCC is happy if two methods share a single field, and does not improve further if the number of shared fields increases. Hence pulling up a field to a superclass can cause SCOM to deteriorate, while TCC remains unaffected.
- If a method does not access any fields, SCOM is impervious to it being moved to another class.

Figure 5 presents a heat map of the amount of metric change that occurs during IRD across all applications. It is apparent that TCC and SCOM have the smallest impact on the other metrics. For TCC this happens as the majority of refactorings that improved this metric had no effect on the other metrics. On the other hand, this happens for SCOM because it is a highly inert metric as shown in Fig. 4.

A striking feature of this heat map is that CC has the maximum effect on reducing the value of the other metrics, especially TCC. A closer look at the results for IRD(CC, TCC) showed that in spite of this strong negative effect that CC has on TCC, the number of refactorings that decreased TCC were about half the number of refactorings that decreased the other metrics. To find the reason behind this apparent anomaly, the refactorings themselves were investigated in more detail.

A closer look at the results shows that the most common refactoring type applied in IRD(CC, TCC) was Pull Up Field. We observe from Fig. 3 that in both CC and TCC the number of fields in the class is not included in the denominator part of these metric, while it has a direct negative effect on the other metrics. Therefore, when an unrelated or less cohesive field is pulled up to a superclass, it has no effect on the CC or the TCC value of the superclass. However, while CC measures only *direct* field usage, TCC measures both *direct*

		Decreasing or not changing					
		LSCC	CC	SCOM	TCC	LCOM5	Total
Increasing	LSCC		0.025	0.024	0.051	0.049	0.149
	CC	0.052		0.125	0.273	0.111	0.561
	SCOM	0.005	0.017		0.051	0.026	0.1
	TCC	0.006	0.012	0.011		0.025	0.053
	LCOM5	0.014	0.015	0.023	0.071		0.122

Fig. 5 The amount of change in metric value during IRD across all applications. The metric on the left is leading and so can only increase, while the metric along the top decreases or remains the same. The value in each cell indicates the total amount of metric difference aggregated across all applications

and *indirect* field usage. Consider a case where the only source of cohesion between a set of methods $m_1 ... m_n$ is their shared indirect access to a field f. To CC, the methods lack cohesion entirely while TCC acknowledges their cohesion through f. In this case, moving the field f to the superclass will not affect CC but will dramatically reduce the value of TCC for the class and hence has a negative overall effect on the cohesion of the program. From this analysis we can make the following observations about the CC and TCC metrics:

- Both CC and TCC ignore the number of fields in a class.
- CC does not take indirect access to a field into account in calculating cohesion.
- The *number* of attributes two methods share is not taken into account in calculating TCC

The authors regard these issues as contentious and worthy of debate. Adding unrelated and unused fields to a class is likely to be viewed by some software developers as weakening the cohesion of the class. With regard to the second point, some developers prefer to access fields through accessor methods even from within the class itself, as this promotes encapsulation, and indeed the *property* feature of the C# language specifically promotes this practice. Code that was written to follow this guideline would be regarded as extremely uncohesive by the CC metric. Finally, it appears intuitive to the authors that the more fields two methods share, the more cohesive the methods are, but TCC does not embody this intuition. Note that our experimental approach can highlight these anomalies; we are not trying to resolve them in this work.

Overall IRA and IRD provide another perspective on how the cohesion metrics compare and enable us to pinpoint further areas of conflict that can be analysed qualitatively to better understand the causes of this conflict.We now continue to consider another set of pairwise metric experiments, Gap Opening Refactoring (GOR) and Gap Closing Refactoring (GCR).

5.2 Gap Opening Refactoring (GOR) and Gap Closing Refactoring (GCR)

Software engineering decision makers are typically advised to record the values of several metrics in making decisions about interventions in software engineering processes (Shepperd 1995). If the initial values reported by two metrics differ significantly this might potentially be confusing or provide little information of value to the decision maker. For example, if one metric reports the cohesion of the system as 0.2, while another reports 0.9, the decision maker may conclude that no information can be gained from either metric.

In such situations, it may be useful to know how much the two metrics can be forced to agree about the value for the software system under investigation. That is, it would be interesting attempt to close the gap between the two metrics using automated refactoring. Since refactoring is behaviour preserving, it does not reduce the complexity of the software in a fundamental way[1]; rather the before and after versions represent slightly different designs for the essentially the same program. If the refactoring reduces the gap between the two metrics, we can assume that this difference represents metric imprecision; the unwanted 'noise sensitivity' produced by syntactic differences in the two versions of the program. We can

[1]By 'complexity' we mean the complexity of the *design* of the software. This complexity is a combination of the complexity of the domain that the software is modelling and the additional complexity of the implementation itself. The domain complexity cannot be changed by refactoring, but the implementation complexity can be increased arbitrarily or reduced to some degree.

also perform the dual operation (gap opening) of course. Both opening and closing operations seek to exploit disagreements between metrics that are due to apparently superficial syntactic details.

To give a concrete example of such superficial details, we discovered in Section 5.1 that TCC and CC are unaffected by the addition of unused fields to a class, whereas LCOM5, LSCC and SCOM view this as detrimental to the cohesion of the class. Thus the superficial detail of adding or removing an unused field to a class will change the gap between TCC/CC and LCOM5/LSCC/SCOM.

Through these two approaches of opening and closing metric gaps we can assess the *extent* to which the metrics could have produced different values for programs that share the same behaviour. Loosely speaking, we can think of this as an informal 'confidence interval' for the possible range of values that the metrics might realistically be expected to take for the program under investigation; they represent the extreme points of values that could be achieved for programs with identical behaviour but different syntax.

Perhaps, after gap opening and closing, we still find that the two metrics exhibit wide disagreement; that there is no overlap between the two 'confidence intervals'. In this case we conclude that the two metrics represent irreconcilable views of the cohesion of the program. However, should we find that the two intervals overlap non-trivially, then we may conclude that the apparent extreme disagreement we initially observed between the metric values is not of any consequence. This gap analysis may thereby help the decision maker to decide whether the apparent disagreement between metrics is germane and inherent to the system under investigation or whether it is merely the result of superficial noise.

It may appear mathematically suspect to compute the difference between two different metrics. However the pair of metrics in question are both cohesion metrics that purport to measure the same property, and the entity that is being measured is the same in both cases. Thus, although the difference in metric values is not meaningful in itself, the extent to which it can be manipulated by refactoring does yield insight into the metrics.

In Section 4.1.2 we stated that two metrics *agree* with each other after a refactoring if either both metric values increase, both decrease, or both stay the same. From this it can seen that *disagreement* occurs when one metric increases or decreases, while the other does the opposite or stays the same. In this section we are investigating disagreement further, but take a broader view of disagreement by considering how the *gap* between the two metrics changes as a result of refactoring. When two metrics disagree, the gap between them widens. However the gap can also widen when they agree or disagree. This occurs when both increase, and the metric with the higher value increases more, or when the both decrease, and the metric with the lower value decreases more. In a similar way, the gap can also narrow when they agree. Therefore the notion of gap opening and closing actually crosscuts the notions of agreement and disagreement as defined in Section 4.

In this section we investigate, for a pair of metrics, to what extent the gap between them can be *widened* by refactoring (Gap Opening Refactoring (GOR)) and to what extent the gap between them can be *narrowed* by refactoring (Gap Closing Refactoring (GCR)).

While GOR and GCR are, in a sense, opposites of one another, they are both measures of a type of difference between the metrics. If two cohesion metrics represent very similar interpretations of cohesion, it would be hard to either increase or decrease the gap between them by refactoring. In fact, GOR and GCR measure the same aspect of the relationship between two metrics as can be seen from the following example. Let P be a program and P' the program that results from the application of the refactoring sequence $R_1 \ldots R_n$, where

each $R_k, k \in [1..n]$ increases the gap between the metrics m_1 and m_2, thus yielding a large value for GOR. All refactorings can be reversed, and the reversal is itself a refactoring. Assuming R'_k to be the reverse refactoring of R_k, we see that starting from program P', each refactoring in the sequence $R'_n ... R'_1$ will decrease the gap between the two metrics m_1 and m_2, thus yielding a large value for GCR. Thus we see that the summation of GOR and GCR tells us something about the metrics, but the exact *distribution* of this value between GOR and GCR depends on the starting program, and not on the metrics themselves.

In the experiments in this section we again compare each pair of metrics from the set of five metrics defined in Fig. 3, thus requiring 10 comparisons in all. Each experiment was performed on the 10 software applications described in Table 1. In each case, we start with the initial application and apply the algorithm depicted in Fig. 2. In the case of GOR, the fitness function accepts a refactoring if and only if it increases the gap between the metrics being compared; likewise for GCR the fitness function accepts a refactoring if and only if it decreases the gap between the metrics being compared. A single run was performed in each case. The refactoring process was usually allowed to run until no further refactorings could be found to open/close the gap, though in the cases of mxGraph, ArtofIllusion, and HTMLUnit the process was terminated after one week.

The results are presented in Fig. 6. In all, 25,682 refactorings were performed that increased the gap, while 19,739 were found that closed the gap. On average then, 4,542 refactorings were found per application that opened or closed the gap between the metrics.

The results for LCOM5 appear striking, but this is not in fact significant. An LCOM5 value of 1 reflects the worst cohesion, while for the other metrics a value of 1 reflects perfect cohesion. Hence the gap between LCOM5 and any other metric is increased by a refactoring that decreases cohesion for both metrics. Such refactorings are easy to find, and this results in the large number of refactorings whenever LCOM5 is involved. Furthermore, the nature of LCOM5 is such that the average initial LCOM5 value for the software applications studied was 0.93, whereas the average value across all the other metrics was 0.14. So the initial gap between LCOM5 and the other metrics tends to be large, and only a small percentage increase/decrease is achieved in spite of the large number of refactorings applied.

Returning to Fig. 6, for some metric pairs it is clear that the gap between the metrics can be opened/closed considerably. In five particular instances, GCR completely removed the gap between the metrics, namely:

- LSCC and CC for XOM and JHotDraw;
- CC and SCOM for mxGraph and HTMLUnit;
- LSCC and SCOM for HTMLUnit.

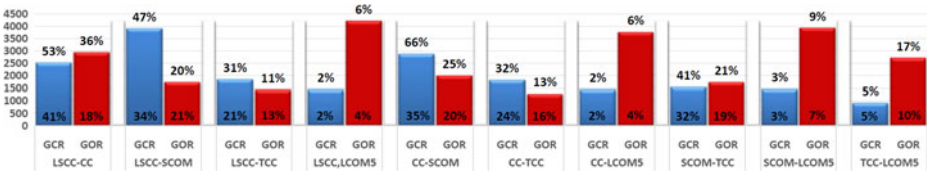

Fig. 6 GCR and GOR for all applications. The percentage above each bar shows the extent to which the gap was closed/opened by GCR/GOR respectively; the y-axis shows the total number of refactorings applied; the standard deviation is shown at the base of each bar

So, for example, a software engineer using cohesion metrics to assess the quality of JHotDraw could safely ignore the difference in metric values between LSCC and CC and attribute this difference to inconsequential differences. On the other hand, the gap between LSCC and TCC for JHotDraw could be reduced by 57 %, which suggests that these two metrics are telling a different story about the cohesion of the JHotDraw application. It is notable that the most obdurate metric difference was found between LSCC and CC on the JRDF application, where GCR could achieve only 0.008 % gap closure. So the same two metrics (LSCC and TCC) can produce a gap for one application (JHotDraw) that GCR can refactor away to zero, while on another application (JRDF) they produce a gap that is highly resistant to GCR.

The results in this section confirm the high level of metric conflict that was already observed in previous experiments. More importantly, GOR/GCR also provide a practical technique for software engineer to determine if the difference between metric values on a particular application are of significance or not.

6 Investigation III: to Inherit or Not to Inherit

Another application of our framework is to investigate various claims and beliefs that exist about metrics but that hitherto could not be experimentally tested. In this section, we test one particular example, the issue of whether or not inherited attributes should be considered in a metric definition. This section illustrates the way in which other researchers can use our search-based exploration of metric volatility spaces to systematically and experimentally investigate open questions in the software metric literature.

In the paper that introduces the TCC metric, Bieman and Kang (1995), the authors observe that inherited methods and fields may or may not be included in calculating a metric value, but make no further observation on whether or not they should be included. In Al-Dallal and Briand (2012) where the LSCC metric is introduced, the authors evaluate whether or not inherited methods should be considered in calculating cohesion in terms of how well it serves as a predictor of a fault existing in a class, but do not consider further the impact of including inheritance in the definition of a cohesion metric. If inheritance is taken into account, then the cohesion of a class is calculated as if all inherited methods and fields were part of the class as well. In the view of the authors of this present paper, this may be a critical issue. Two methods in a class might appear to be unrelated when inheritance is omitted, but if they both access the same inherited methods or fields they might in fact be very cohesive[2] Hence we consider two forms of each of the metrics under consideration, the normal, 'local' form and the 'inherited' form, which we denote by appending the subscript 'i', e.g. LSCC_i.

To investigate the effect of inheritance on the metric definitions, we performed an experiment to compare 10 metrics: the five we have already considered CC, LCOM5, LSCC, SCOM, TCC and their 'inherited' counterparts, CC_i, LCOM5_i, LSCC_i, SCOM_i and TCC_i. Each metric pair (90 in all) was evaluated on one software application, namely JHotDraw. We use only one application for these 90 experiments, so as to make the results manageable. We chose JHotDraw as it proved in Section 4 to be the application that Code-Imp found

[2]The Template Method design pattern (Gamma et al. 1995) is an example of this. The subclasses contain several apparently unrelated methods. However, it is the inherited template method itself that provides the glue that makes these methods cohesive.

easiest to refactor, and because its rich inheritance hierarchy makes it suitable for this type of investigation.

For each metric pair (m_1, m_2), two experiments were performed, one where a refactoring is accepted only when it leads to an increase in m_1 and the other where a refactoring is accepted only when it leads to an increase in m_2. In each case, the value of the dependent metric is measured after each refactoring and what is of interest is how the changes in the two metrics correlate with each other during the refactoring sequence.

This experiments in this section involved a total of 10,505 refactorings, broken down as depicted in Fig. 7. The results of these experiments are presented in Fig. 8. If the regular form of a metric and its 'inherited' form embodied similar interpretations of cohesion, we would expect each odd-numbered row (the regular form) in the figure to be similar to the one below it (the 'inherited' form) and we would expect to see strong positive correlation when the normal metric form is played against its 'inherited' counterpart. However, it is evident from the data that in fact the normal and 'inherited' forms behave in very different ways and appear to be completely different metrics. For example, when TCC leads the search, LSCC correlates strongly with it, but when TCC_i leads the search, $LSCC_i$ exhibits a strong negative correlation; when SCOM leads, $LCOM5_i$ correlates strongly but when $SCOM_i$ leads, $LCOM5_i$ exhibits strong negative correlation. There are many examples of these anomalies across Fig. 8. Formal statistical analysis is unnecessary to reject the popular notion that the inclusion or non-inclusion of inheritance in a metric definition is a matter of small import.

6.1 Qualitative Analysis

In this section we examine some of the refactorings that caused the normal and 'inherited' forms of the metrics to exhibit such different behaviour, in an effort to gain further insight into the differences between the metrics themselves. For reasons of space, we limit ourselves to considering three metric pairs, namely (LSCC, $LSCC_i$), (TCC_i, $LSCC_i$) and (LSCC, TCC).

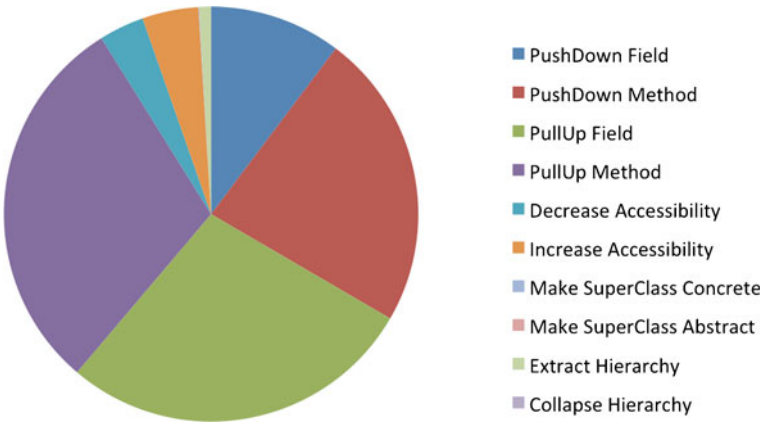

Fig. 7 Breakdown of the refactorings applied to JHotDraw during investigation III

	Measure →									
Improve↓	LSCC	LSCCi	CC	CCi	SCOM	SCOMi	TCC	TCCi	LCOM5	LCOM5i
LSCC		-0.871	0.836	-0.623	0.977	-0.187	0.997	0.832	-0.632	0.898
LSCCi	0.817			0.591	0.826	0.824	0.990	0.997	0.930	-0.945
CC	-0.895	-0.995		0.793	-0.749	-0.890	-0.667	-0.959	0.924	0.979
CCi	-0.983	-0.826	0.983		-0.768	-0.997	-0.880	0.594	0.806	0.803
SCOM	0.989	-0.644	0.427	0.064		0.755	0.980	0.838	-0.727	0.943
SCOMi	0.985		-0.396	0.924	1		0.986	0.751	-0.867	-0.881
TCC	0.989	0.649	0.940	0.772	0.782	0.916		0.707	0.697	0.930
TCCi	0.837	-0.894	0.657	0.848	-0.507	-0.974	0.853		0.981	0.987
LCOM5	-0.230	0.812	-0.974	-0.725	-0.972	-0.967	-0.396	0.659		-0.797
LCOM5i	0.940	-0.914	-0.713	-0.995	0.858	-0.870	0.900	-0.709	-0.919	

Fig. 8 Spearman rank correlation between all metrics across all refactorings applied to JHotDraw. Only cases where a significance value of $<= 0.05$ was found are shown

6.1.1 Comparing LSCC and LSCC$_i$

When LSCC leads the search, LSCC$_i$, exhibits strong negative correlation (-0.871). On examining the refactorings, we see that this negative correlation can be attributed mainly to Pull Up Method and especially Pull Up Field refactorings. Moving a less cohesive method, or especially a less cohesive field, from a subclass to its superclass can improve the LSCC cohesion value of the subclass. While these refactorings typically reduce the cohesion of the superclass, overall they can improve the cohesion of the program if the subclass is more highly weighted than the superclass. However, for LSCC$_i$, these refactorings reduce the cohesion of the superclass but have no effect on LSCC$_i$ for the subclass as the moved fields and methods are still accessible in the subclass. In addition, if the Increase Accessibility refactoring is also called on the moved field and method, e.g. to change the accessibility from private to protected[3], then it can also result in a reduction in cohesion for all children of the subclass. This explains why this type of refactoring can improve LSCC while causing a reduction in LSCC$_i$.

On the other hand, it was observed that in the LSCC versus LSCC$_i$ experiments, certain refactorings such as Push Down Method have a positive effect on both metrics. Moving an uncohesive method to a subclass where it is used improves the cohesion of the superclass for both metrics, and improves the cohesion of the subclass for LSCC, but does not change LSCC$_i$ for the subclass.

6.1.2 Comparing TCC$_i$ and LSCC$_i$

When TCC$_i$ leads the search, LSCC$_i$ exhibits strong negative correlation (-0.894). The most striking feature here is that the Pull Up Field refactoring has a negative impact on LSCC$_i$ in

[3]In this case the moved element is weakly cohesive within the subclass, but it is used by at least one method. Therefore its accessibility in the superclass must be changed to non-private to remain accessible in the subclass.

every case. The negative impact occurs because a field is moved to a superclass where it has no interaction which reduces $LSCC_i$ for that class. TCC_i favours this refactoring because as part of pulling a private field up to a superclass, it must be made protected, and this causes more interaction between protected methods that use the field in the hierarchy structure. This use of Pull Up Field in this case does not truly improve cohesion, so it is a strength of $LSCC_i$ that it would not recommend it.

Another area of conflict is the negative effect Push Down Method has on $LSCC_i$ in a number of refactorings. On inspecting these refactorings, we learn that TCC_i always prefers a method to reside in a class where it is used and access the fields it needs in its superclass (where they cannot be private of course), rather than reside in the superclass. However, $LSCC_i$ places more emphasis on keeping fields private, so it frequently prefers a method to stay in the class of the fields it uses except where the method is used by majority of the subclasses.

6.1.3 Comparing LSCC and TCC

It is striking how closely LSCC and TCC correlate with each other in these experiments. A closer look at the refactorings involved shows that the main refactoring with a positive effect on both TCC and LSCC is Pull Up Method. A method that is weakly cohesive in its own class is pulled up to its direct superclass where it is more cohesive, hence improving the overall cohesion of the program for both TCC and LSCC. This is likely to be a good refactoring if the superclass has only one child. However, if the superclass has more than one child then the negative impact of this refactoring on the other children may be a factor. Note that the normal forms of the metrics do not register the negative effects of this refactoring, but $LSCC_i$ and $SCOM_i$ can indeed detect it, which illustrates the benefit of including inherited attributes in a metrics definition.

In spite of the close correlation between LSCC and TCC, some conflict is observed and, given its rarity, this is interesting to examine further. One period of disagreement occurs during a sequence of Pull Up Field refactorings where the target class has no fields. TCC is undefined for a class with no fields, so moving a field to such a class appears to reduce cohesion by adding a class with zero cohesion to the program. On the other hand, we learn from this example that LSCC prefers to move a field that is loosely associated with a class (e.g. used directly or indirectly by only one method) to its superclass, if that superclass has a zero LSCC measure (no two methods access the same field). In practice, this would be viewed as a detrimental refactoring, so we have uncovered a issue with the LSCC metric that it would reward such a refactoring.

6.2 Summary

In this section we explored the popular notion that including inherited methods and fields in the computation of a metric for a class is largely a matter of taste, and found the contrary to be true: when inherited methods and fields are included in the computation of a metric, the metric is liable to yield very different values even to the extent of exhibiting strong negative correlation with the original metric. While this is an interesting result in itself, the goal of this section is broader than just this issue. Other claims of a similar nature that are made about software metrics, e.g. that improving cohesion should also decrease coupling, can be evaluated experimentally using search-based refactoring as we have done here.

7 Related Work

In this section we review related work in Search-Based Refactoring (Section 7.1) and Software Metrics (Section 7.2).

7.1 Search-Based Refactoring

Search-based refactoring is fully automated refactoring driven by metaheuristic search and guided by software quality metrics, as introduced by O'Keeffe and Ó Cinnéide (2003). Existing work in this area uses either a 'direct' or an 'indirect' approach. In the direct approach the refactoring steps are applied directly to the program, denoting moves from the current program to a near neighbour in the search space. Early examples of the direct approach are the works by Williams (1998) and Nisbet (1998) who addressed the parallelization problem. More recently, O'Keeffe and Ó Cinnéide (2008a, b) applied the direct approach to the problem of automating design improvement. They used a collection of 12 metrics to measure the improvements achieved when methods are moved between classes, new classes created and associations between classes changed.

In the indirect approach, the program is indirectly optimised through the optimisation of the sequence of transformations to apply to the program. In this approach fitness is computed by applying the sequence of transformations to the program in question and measuring the improvement in the metrics of interest. The first authors to use search in this way were Cooper et al. (1999), who used biased random sampling to search a space of high-level whole-program transformations for compiler optimisation. Also following the indirect approach, Fatiregun et al. (2004, 2005) showed how search-based transformations could be used to reduce code size and construct amorphous program slices. The work of Kessentini et al. also uses the indirect approach extensively to improve a software system by improving software quality metrics and reducing instances of code smells (Ouni et al. 2012, 2013a,; Mkaouer et al. 2014). The same authors also use optimisation techniques and code development history to guide a search towards an optimal refactoring sequence for minimisation of code smell instances (Ouni et al. 2013b).

Ghaith and Ó Cinnéide (2012) used search-based refactoring to improve the security of code. An empirical study of a banking system showed improvements of over 15 % in program security. The fitness function used was based on a set of security metrics, although the authors point to certain weaknesses in the security metrics themselves as a drawback of the study. Mahouachi et al. (2013) describe an approach to automate detection of refactorings in source code using structural information. Open-source systems were used as the empirical basis and 90 % precision and recall were observed. Ghannem et al. (2013) use an Interactive Genetic Algorithm (GA) which interacts with users while allowing feedback to a normal GA. The implemented tool was used to suggest sequences of refactorings which could be applied to models in the form of class diagrams. The extent to which an interactive approach could be applied and validation of the correctness of suggested refactorings were explored and both showed promise. Hemati Moghadam and Ó Cinnéide (2012) present an approach which refactored a program based on its desired design and source code. Open-source Java was used as an empirical basis and results from the study showed that the original program could be refactored to that design with 90 % accuracy.

Seng et al. (2006) propose an indirect search-based technique that uses a genetic algorithm over refactoring sequences. In contrast to O'Keeffe M and Ó Cinnéide M (2006),

their fitness function is based on well-known measures of coupling between program components. Both these approaches used weighted-sum to combine metrics into a fitness function, which is of practical value but is a questionable operation on ordinal metric values. A solution to the problem of combining ordinal metrics was presented by Harman and Tratt (2007), who introduced the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics into a fitness function and demonstrated that it has several advantages over the weighted-sum approach.

The work of Sahraoui et al. (2000) has some similarities to ours, notably their premise that semi-automated refactoring can improve metrics. Their approach attempts to gain insight into the refactorings that are chosen to improve a chosen metric. Our approach is the reverse of this: we use refactorings to gain insights into several metrics. A similar approach is that of Chaparro et al. (2014), who developed a technique called RIPE (Refactoring Impact Prediction) that tries to support the developer by predicting what impact a refactoring will have on the metric values for a given software application. They achieve some success with this, but the fact that the metric change cannot be exactly calculated lends support to the approach we have taken in this paper.

Otero et al. (2010) 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 (2010) use genetic programming to drive a search-based refactoring process that aims to introduce design patterns. Ó Cinnéide et al. (2011) use a search-based refactoring approach to try to improve program testability. Kilic et al. (2011) explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Oliveira Barros de and Almeida Farzat de (2013) suggest that while current search-based approaches may produce complex structures in the design process, they nonetheless allow a critique of software metrics and lead to new avenues of research, in particular that of refactoring. Finally, Tsantalis and Chatzigeorgiou (2011) used a semi-automated process for identifying refactoring opportunities based on code smells; the system's history was used as a basis for the study.

7.2 Analysis of Software Metrics

One criticism that is levelled at the use of software metrics is that they often fail to measure what they purport to measure (Fenton and Pfleeger 1996). This has led to a proliferation of software metrics (Fenton and Neil 2000), many of which attempt to measure the same aspect(s) of code. It is not surprising then that several studies have attempted to compare software metrics to better understand their similarities and differences. Work by Stuckman et al. (2013) for example, used synthetic defect datasets to assess the performance of metrics, complemented by a formal mathematical model. One of the results of the study was that a relatively small set of source code metrics conveyed the same information as a larger set.

In this section, we focus on studies that have analysed cohesion metrics. The overriding problem with cohesion (and its measurement) has been that, unlike coupling, any metric claiming to measure cohesion is relatively subjective and open to interpretation (Counsell et al. 2005). Most cohesion measures have focussed on the distribution of attributes in the methods of a class (and variations thereof). However, nuances of different object-oriented languages and the fact that the distribution of attributes can make it impossible to calculate cohesion metrics, means that no single, agreed cohesion metric exists.

The LCOM metric has been subject to detailed scrutiny (Briand et al. 1998) and revised several times to account for idiosyncrasies in its calculation. Comparisons between LCOM and other proposed cohesion metrics are a common feature of empirical studies (Bieman and Kang 1995; Bansiya et al. 1999; Counsell et al. 2006; Al Dallal 2010; Al-Dallal and Briand 2010, 2012). Most newly-proposed cohesion-based metrics have attempted to improve upon previous metrics by forming a link between low cohesion and high fault-proneness (Al-Dallal and Briand 2010, 2012) or intuitive notions of high cohesion and subjective developer views of what constitutes high cohesion (Bansiya et al. 1999); others have tried to demonstrate a theoretical improvement (Counsell et al. 2006; Al Dallal 2010). Comparison of cohesion metrics has been a consistent topic for research (Succi et al. 2005; Joshi and Joshi 2010; Kaur and Singh 2010). For example, the Cohesion Amongst the Methods of a Class (CAMC) metric (Bansiya et al. 1999) extends the LCOM metric by including the `self` property in C++ in its calculation, and has been validated against developer opinion.

Al Dallal (2010) investigated the relationship between their proposed metric, Low-Level Similarity-Based Class Cohesion (LSCC), and eleven other low-level cohesion metrics in terms of correlation and ability to predict faults. Based on correlation studies they concluded that LSCC captures a cohesion measurement dimension of its own. Four open source Java applications consisting of 2,035 classes and over 200KLOC were used as a basis of their study.

The same authors also propose a high-level design class cohesion metric - the Similarity-Based Class Cohesion Metric (SCC) (Al-Dallal and Briand 2010). They explored correlations among SCC and ten other cohesion metrics. The result showed a moderate correlation between SCC and the other metrics scrutinised, while some much stronger correlations between the SCC and other previously defined cohesion metrics including LCOM were observed (Bansiya et al. 1999; Chidamber and Kemerer 1994; Counsell et al. 2006). At the very least, the SCC metric could be used as a surrogate for LCOM. Al-Dallal (2013) explored cohesion metrics from the perspective of transitive relationships between attributes; a method is related to the attributes it references, but also to the attributes referenced by the methods that it invokes. Current cohesion metrics fail to consider these relationships and do not therefore capture cohesion in its true sense.

Counsell et al. (2006) proposed a new metric, the Normalized Hamming Distance metric (NHD), and found evidence that NHD is a better cohesion metric than CAMC. Their empirical data, obtained from three C++ applications, showed a strong negative correlation between NHD and other metrics. This contrasts with a more recent study by Kaur and Singh (2010) who explored the relationship between NHD, SNHD (Counsell et al. 2006) and CAMC. They observed that class size was a confounding factor in the computation of both CAMC and NHD.

Alshayeb (2009b) observed that refactoring had a positive effect on several cohesion metrics in his study of open source software. However, in later work he reported that this effect was not necessarily positive on other external software quality attributes such as reusability, understandability, maintainability, testability and adaptability (Alshayeb 2009a). An information-theoretic approach to measuring cohesion was proposed by Allen et al. (2001) and while this represented a fresh approach to cohesion measurement, their metric is subject to the same criticisms as previous metrics.

Veerappa and Harrison (2013) used automated refactoring techniques to investigate the behaviour of coupling metrics during refactorings. Eight open source Java projects were used as the empirical basis. Results showed that in contrast to cohesion metrics, coupling metrics were less likely to conflict with each other.

In terms of metrics for uncovering new facets of cohesion and as a basis for fault prediction, several studies have made a contribution. Marcus et al. (2008) proposed a new metric for measuring cohesion using information in source code such as comments; the study drew heavily on the areas of psychology and linguistics. The Conceptual Cohesion of Classes metric (C3) highlighted properties of code orthogonal to that previously shown by cohesion metrics; when combined with existing structural cohesion metrics, C3 gave better defect prediction capability *vis-a-vis* structural cohesion metrics on their own (three open-source systems were used as the empirical basis). In a similar way, Liu et al. (2009) also investigate a new measure for cohesion using information theory concepts (the Maximal Weighted Entropy metric) based on topics embedded in code (i.e., comments and identifiers). The measure was shown to highlight new facets of cohesion and its effectiveness as a fault prediction mechanism demonstrated; open-source (Mozilla) was used as a basis of the study. Two new cohesion and coupling metrics for fault prediction were proposed by Ujhazi et al. (2010). The Mozilla open-source system was again used a basis of their empirical study and machine learning techniques used to assess predictive capability of these metrics. Combining these two new metrics with other metrics (from a code quality framework) showed an improvement in defect prediction capability.

In recent work, Meneely et al. (2013) performed a meta-analysis of software metric literature to uncover the validation criteria that authors use in evaluating a new metric. In all, they identified a very diverse set of 47 such criteria. A number of these criteria can be assessed using our search-based refactoring approach, e.g.

- *Actionability*: A metric that is found to be highly inert (see Section 4.1.1) will not be actionable, i.e. it will be hard to refactor the program to improve it for this metric;
- *Construct validity*: While search-based refactoring cannot assess construct validity directly, by showing that several cohesion metrics conflict heavily it has been demonstrated that they cannot all have good construct validity.
- *Nonexploitability*: A metric exhibits nonexploitability if developers cannot manipulate a metric to obtain desired results (Cavano and McCall 1978). A metric that is found to be highly volatile using search-based refactoring is likely to be easily exploitable by developers.

However the cross-validation that our approach can achieve is not represented in any of the validation criteria defined in this work.

A study by Kitchenham (2010) also involved a meta-analysis of software metrics literature to identify trends in influential software metrics papers and assess the possibility of using secondary studies to integrate research results. Her investigations did not uncover any papers that use an experimental approach to metrics assessment, as we do. One of her criticisms is that some authors have empirically studied metrics that are already known to be theoretically unsound. We avoided this by choosing a revised version of the original and unsound metric LCOM, namely LCOM5, to use in our experiments.

These studies have created a deeper understanding of software metrics and have shown that metrics with a similar intent do not necessarily provide similar results. However, understanding the underlying characteristics of a metric is just a first step in determining their usefulness. When metrics are applied to real-world programs, they can yield results that were not obvious from their definitions, in the same way that a program under test can produce results the developer never intended or even imagined. The approach detailed in this paper takes the next step by quantifying the extent of conflict between metrics to understand them better and to pinpoint the root cause of the conflict.

8 Threats to Validity

There are four principal types of threat that can affect the validity of our experiments, namely internal validity, construct validity, external validity and conclusion validity (Wohlin et al. 2012). We consider each of these in the following paragraphs.

Internal validity in concerned with the relationship between the treatment and its outcome. This must be a causal relationship, i.e. the outcome must be as a result of the applied treatment and must not be as a result of some other factor that we have not taken into account. In our experiments, the treatment is the application of a refactoring and the outcome is the change in metric values for the program being refactored. As the software is not being changed in any other way, and software is not open to the myriad of biases that affect studies with human subjects, it is trivial to establish that the changes we observe in the metric values are attributable to the refactoring that has been applied to the program.

Another concern here is the correctness of refactoring transformations themselves. It has been established that many current refactoring tools contain bugs (Schäfer et al. 2012; Gligoric et al. 2013) and our refactoring implementations, although heavily tested, cannot be expected to be perfect. However, for the purposes of this paper, refactoring correctness is not of paramount importance. Even if a refactoring is erroneously implemented and fails to preserve program behaviour in some circumstances, it is still a program transformation and how the metrics change is response to this transformation is of interest.

Construct validity is concerned with the relationship between theory and observation, both in the treatment and the outcome. In terms of the treatment, we wish to apply small, controlled changes to a program that do not change it radically and in some sense leave it 'the same'. Refactorings fit this description well. In a sequence of programs linked by individual refactorings, each program exhibits the same external behaviour, and, as each program is similar to its neighbour it also has a similar internal complexity in some sense (Section 5.2 elaborates on what we mean by 'complexity' in this context). Other non-refactoring program transformations may be suitable for this purpose as well, but this does not threaten the suitability of refactorings for this purpose.

The issue of construct validity regarding the outcome is a different matter. The outcome in our experiments is the changes to the cohesion metrics under study. Formally speaking, we have achieved construct validity if the cohesion metrics we employ truly represent cohesion. Each cohesion metric we study has been used in software practice and has been the subject of other research studies, and so can be claimed to represent well the cohesion construct. However, that is to beg the very question we are investigating, because in our experiments we are in fact testing the construct validity of the cohesion metrics under study. It is precisely by seeking areas where these metrics conflict that we are able to argue that they do not represent the same construct and hence represent different, and indeed conflicting, theoretical notions of cohesion.

External validity refers to the generalisability of our findings, and in this context there are a number of limitations to this work that should be clarified. We analyse a broad range of software systems, but they are all open source systems and it may be the case that closed source systems would yield different results. It is also possible that the open source applications and the set of cohesion metrics we chose for our experiments happen by chance to display considerable conflict, and that this would not generalise to other combinations of applications and metrics. To ameliorate these threats, the 10 software applications we used were chosen at random and the five metrics we used were selected randomly from popular

cohesion metrics and we had no prior knowledge that any conflict would occur, i.e. we did not in any sense 'fish' for conflicting examples.

Our results are also affected by the choice of refactoring types that we made. For example, refactorings that break up methods (Extract Method etc.) could well lead to very different results. However, this cannot of course reduce the amount of conflict observed using the set of refactorings we employed, and there is currently no agreed complete set of refactoring types. In this paper, we used a hill-climbing variant as the metaheurstic search mechanism. We could be criticised for not applying other search-based techniques, such as simulated annealing or genetic algorithms. However, the focus of this paper is primarily a comparison of metrics and an analysis thereof, rather than a comparison of search techniques. It is possible that application of these techniques would lead to new insights into the results. However, this would be a largely orthogonal study and, as such, is a topic for future work.

Our study has explored a subset of metrics designed to measure program cohesion and has cross-compared those metrics. Many other cohesion metrics exist and there is the possibility that other class features, if incorporated into a metric, may provide new insights into cohesion measurement that our study has not revealed. However, the approach adopted in this paper provides a rigorous means of comparing and contrasting any set of metrics (e.g. for coupling (Veerappa and Harrison 2013)) and this is of value to the metrics community in understanding inter-metric relationships and, ultimately, their true value to practitioners.

Finally, conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We used Spearman rank correlation to measure the relationship between the metric changes in Investigations I and III. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so it is safe to use in this context.

9 Conclusions and Future Work

In this paper we use search-based refactoring for a novel purpose: to discover relationships between software metrics. By using a variety of search techniques (semi-random search, refactoring for increased metric agreement/disagreement and refactoring to increase/decrease the gap between a pair of metrics) guided by a number of cohesion metrics, we were able to make experimental assessments of the metrics. In areas of direct conflict between metrics, we examined further the refactorings that caused the conflict in order to learn more about nature of the conflict and gain further qualitative insight into the differences between the metrics. The new information gained about software metrics could not be gleaned using any of the existing approaches to metrics assessment.

In our initial study of 300KLOC of open source software we found that the cohesion metrics LSCC, TCC, CC, SCOM and LCOM5 agreed with each other in only 45 % of the refactorings applied. In 17 % of cases dissonance was observed (one metric changing while the other remains static) and in 38 % of cases the metrics were found to be in direct conflict (one metric improving while the other disimproves). This high percentage of conflict reveals an important and surprising feature of cohesion metrics: they not only embody *different* notions of cohesion, they embody *conflicting* notions of cohesion. This key result means that the notion of combining current cohesion metrics into a single, unifying cohesion metric is an impossibility.

Through the use of Iterative Refactoring Agreement/Disagreement (IRA and IRD) we found that LSCC is the best representative of the set of metrics we investigated while SCOM is the least representative. A closer examination of CC and TCC revealed problems with both metrics in terms of how field access within the class affects the metric value.

To address the practical problem where a software engineer has to interpret several metrics with widely-varying values, we introduced the idea of Gap Opening Refactoring (GOR) and Gap Closing Refactoring (GCR). Using these techniques we can estimate if the difference between two metric values is due to an inherent difference between the metrics, or something more superficial. In experiments we found extreme cases where GCR could reduce the gap between metrics to zero (indicating metrics that are only superficially different) and cases where the reduction achievable was as little as 0.008 % (indicating fundamental metric difference).

The refactoring approach proposed in this paper can also be used to investigate claims that are made regarding software metrics. To illustrate this, we explored the decision as to whether or not inheritance should be included in the definition of a cohesion metric given that several sources have discussed this issue but found it to be not of major import (Bieman and Kang 1995; Al-Dallal and Briand 2012). Our findings were that, in general, a metric behaves completely differently when inherited attributes are taken into account. In three areas of conflict between LSCC and TCC our analysis of the refactorings led to detailed insights into the differences between these metrics (see Sections 6.1.2 and 6.1.3).

Our goal in this work is not to resolve these issues, but to provide a methodology whereby they can be detected and analysed in order to aid further metrics research. In some cases, software design principles indicate which metric is best in which context. In other cases, the developer can choose which metric best suits their needs. In any given context, GCR and GOR can be used to as an aid to assessing if a given metric conflict is superficial or fundamental.

We claim that our approach can contribute significantly to the ongoing metrics debate. It provides an automated platform upon which metrics can be animated and their areas of agreement and disagreement brought into clear focus, in a way that is not achievable using existing approaches to evaluation. Future work in this area involves a number of themes. Firstly, we have focused on cohesion metrics and their interplay. It would be interesting to examine the trade-offs between changes in these metrics and for example, those that attempt to measure coupling, particularly since low coupling and high cohesion are desirable system features and could be considered as joint goals of system designers and maintainers. Similarly it would be interesting to study how semantic cohesion (measured using e.g. WordNet) compares with structural cohesion under this type of experimentation. Secondly, our refactoring-based experimental approach can be developed further to create a practical, easy-to-use tool that permits metrics researchers and practitioners to experimentally evaluate new or existing metrics. Thirdly, while our work has highlighted metric conflict, we have not attempted to resolve this conflict and determine which metric is the most suitable under which circumstances. Further work to address this issue will have to involve studies that elicit developer opinion. Finally, we have used open source systems in this study; it would be valuable to apply the same analysis to proprietary systems and to compare the results obtained in each case.

Acknowledgments This work was supported, in part, by grants from the Engineering and Physical Sciences Research Council of the UK (EPSRC) - Grant references: EP/E055141/1 and EP/J017515/1, and by Science Foundation Ireland (SFI) grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre.

References

Al Dallal J (2010) Validating object-oriented class cohesion metrics mathematically. In: Proceedings of the 9th international conference on Software engineering, parallel and distributed systems, SEPADS'10, pp 73–77

Al-Dallal J (2013) Incorporating transitive relations in low-level design-based class cohesion measurement. Software: Practice and Experience 43(6):685–704

Al-Dallal J, Briand LC (2010) An object-oriented high-level design-based class cohesion metric. Inf Softw Technol 52(12):1346–1361

Al-Dallal J, Briand LC (2012) A precise method-method interaction-based cohesion metric for object-oriented classes. ACM Trans Softw Eng Methodol (TOSEM) 21(2):8:1–8:34

Allen E, Khoshgoftaar T, Chen Y (2001) Measuring coupling and cohesion of software modules: an information-theory approach. In: Proceedings of the 7th International Software Metrics Symposium, pp 124–134

Alshayeb M (2009a) Empirical investigation of refactoring effect on software quality. Inf Softw Technol 51(9):1319–1326

Alshayeb M (2009b) Refactoring effect on cohesion metrics. In: Proceedings of the International Conference on Computing, Engineering and Information, ICC '09, pp 3–7

Bansiya J, Etzkorn L, Davis C, Li W (1999) A class cohesion metric for object-oriented designs. Journal of Object-Oriented Programming 11(08):47–52

Beck F, Diehl S (2011) On the congruence of modularity and code coupling. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE '11, pp 354–364

Bieman JM, Kang BK (1995) Cohesion and reuse in an object-oriented system. In: Proceedings of the symposium on software reusability, Seattle, Washington, pp 259–262

Bieman JM, Ott LM (1994) Measuring functional cohesion. IEEE Trans Softw Eng 20(8):644–657

Bonja C, Kidanmariam E (2006) Metrics for class cohesion and similarity between methods. In: Proceedings of the 44th annual southeast regional conference. ACM, Florida, pp 91-95

Bouwers E, Deursen Av, Visser J (2013) Evaluating usefulness of software metrics: An industrial experience report. In: Proceedings of the 35th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 921–930

Briand L, Emam KE, Morasca S (1996) On the application of measurement theory in software engineering. Empir Softw Eng 1:61–88

Briand LC, Daly JW, Wüst J (1998) A unified framework for cohesion measurement in object-oriented systems. Empir Softw Eng 3(1):65–117

Cavano JP, McCall JA (1978) A framework for the measurement of software quality. ACM SIGSOFT Software Engineering Notes 3(5):133–139

Chaparro O, Bavota G, Marcus A, Penta MD (2014) On the impact of refactoring operations on code quality metrics. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 456–460

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Cooper KD, Schielke PJ, Subramanian D (1999) Optimizing for reduced code space using genetic algorithms. In: Proceedings of the ACM workshop on languages, compilers and tools for embedded systems, NY, LCTES '99, vol 34.7, pp 1–9

Counsell S, Swift S, Tucker A (2005) Object-oriented cohesion as a surrogate of software comprehension: an empirical study. In: Proceedings of the 5th IEEE international workshop on source code analysis and manipulation, Washington DC, USA, pp 161-172

Counsell S, Swift S, Crampton J (2006) The interpretation and utility of three cohesion metrics for object-oriented design. ACM Trans Softw Eng Methodol (TOSEM) 15(2):123–149

Fatiregun D, Harman M, Hierons R (2004) Evolving transformation sequences using genetic algorithms. In: Proceedings of the 4th IEEE international workshop on source code analysis and manipulation. IEEE computer society press, los alamitos, pp 65–74

Fatiregun D, Harman M, Hierons R (2005) Search-based amorphous slicing. In: Proceedings of the 12th International Working Conference on Reverse Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, WCRE '05, pp 3–12

Fenton NE (1994) Software measurement: a necessary scientific basis. IEEE Trans Softw Eng 20(3):199–206

Fenton NE, Neil M (2000) Software metrics: Roadmap. In: Proceedings of the 22nd Conference on The Future of Software Engineering, ACM, New York, NY, USA, ICSE '00, pp 357–370

Fenton NE, Pfleeger SL (1996) Software metrics - a practical and rigorous approach (2nd. ed.). International Thomson

Fernández L, Peña R (2006) A sensitive metric of class cohesion. Information Theories and Applications 13(1):82–91

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley

Gamma E, Helm R, Johnson RE, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading

Ghaith S, Ó Cinnéide M (2012) Improving software security using search-based refactoring. In: Proceedings of the 4th International Symposium on Search Based Software Engineering, Riva del Garda, Italy, SSBSE'12, pp 121–135

Ghannem A, El-Boussaidi G, Kessentini M (2013) Model refactoring using interactive genetic algorithm. In: Proceedings of the 5th International Symposium on Search Based Software Engineering, St. Petersburg, Russia, SSBSE '13, pp 96–110

Gligoric M, Behrang F, Li Y, Overbey J, Hafiz M, Marinov D (2013) Systematic testing of refactoring engines on real software projects. In: Proceedings of 27th European Conference on Object-Oriented Programming, Montpellier, France, July, 2013, pp 629–653

Gutzmann T et al. (2013) RECODER: a framework for java program analysis and source code transformation. http://sourceforge.net/projects/recoder

Harman M, Clark J (2004) Metrics are fitness functions too. In: Proceedings of the 10th international symposium on metrics. IEEE Computer Society, USA, pp 58–69

Harman M, Tratt L (2007) Pareto optimal search based refactoring at the design level. In: Proceedings of the 9th Conference on Genetic and Evolutionary Computation, GECCO '07, pp 1106–1113

Harman M, Danicic S, Sivagurunathan B, Jones B, Sivagurunathan Y (1995) Cohesion metrics. In: Proceedings of the 8th international quality week, San Francisco, USA, pp 1–14

Hemati Moghadam I, Ó Cinnéide M (2011) Code-Imp: A tool for automated search-based refactoring. In: Proceedings of the 4th Workshop on Refactoring Tools, ACM, New York, NY, USA, WRT '11, pp 41–44

Hemati Moghadam I, Ó Cinnéide M (2012) Automated refactoring using design differencing. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering, Szeged, Hungary, CSMR '12, pp 43–52

Hemati Moghadam I, Ó Cinnéide M (2015) Resolving conflict and dependency in refactoring to a desired design. e-Informatica Softw Eng J 9(1):37–56

Hitz M, Montazeri B (1996) Chidamber and kemerer's metrics suite: a measurement theory perspective. IEEE Trans Softw Eng 22(4):267 –271

Jensen A, Cheng B (2010) On the use of genetic programming for automated refactoring and the introduction of design patterns. In: Proceedings of the 12th conference on genetic and evolutionary computation. ACM, New York, pp 1341–1348

Joshi P, Joshi RK (2010) Quality analysis of object oriented cohesion metrics. In: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology, IEEE Computer Society, pp 319–324

Kaur K, Singh H (2010) Exploring design level class cohesion metrics. J Softw Eng Appl 03(04):384–390

Kemerer C (1995) Software complexity and software maintenance: a survey of empirical research. Ann Softw Eng 1(1):1–22

Kilic H, Koc E, Cereci I (2011) Search-based parallel refactoring using population-based direct approaches. In: Proceedings of the 3rd International Conference on Search Based Software Engineering, Springer-Verlag, Berlin, Heidelberg, SSBSE'11, pp 271–272

Kitchenham B (2010) What's up with software metrics? - a preliminary mapping study. J Syst Softw 83(1):37–51

Lakhotia A (1993) Rule-based approach to computing module cohesion. In: Proceedings of the 15th International Conference on Software Engineering, ICSE'10, pp 35–44

Liu Y, Poshyvanyk D, Ferenc R, Gyimóthy T, Chrisochoides N (2009) Modeling class cohesion as mixtures of latent topics. In: Proceedings of the 25th IEEE International Conference on Software Maintenance, ICSM '00, pp 233–242

Mahouachi R, Kessentini M, Ó Cinnéide M (2013) Search-based refactoring detection using software metrics variation. In: Proceedings of the 5th International Symposium on Search Based Software Engineering, St. Petersburg, Russia, SSBSE '13, pp 126–140

Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans Softw Eng 34(2):287–300

Meneely A, Smith B, Williams L (2013) Validating software metrics: a spectrum of philosophies. ACM Trans Softw Eng Methodol (TOSEM) 21(4):24:1–24:28

Meyers TM, Binkley D (2007) An empirical study of slice-based cohesion and coupling metrics. ACM Trans Softw Eng Methodol (TOSEM) 17(1):1–27

Mkaouer W, Kessentini M, Bechikh S, Deb K, Ó Cinnéide M (2014) Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proceedings of the 29th IEEE International Conference on Automated Software Engineering, IEEE Press, ASE'14, pp 331–336

Nisbet A (1998) GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In: Sloot PMA, Bubak M, Hertzberger LO (eds) Proceedings of the international conference on high-performance computing and networking, vol LNCS 1401. Springer, pp 987–989

Ó Cinnéide M, Boyle D, Hemati Moghadam I (2011) Automated refactoring for testability. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops, Berlin, ICSTW '11, pp 437–443

Ó Cinnéide M, Tratt L, Harman M, Counsell S, Hemati Moghadam I (2012) Experimental assessment of software metrics using automated refactoring. In: Proceedings of the 6th ACM-IEEE ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, New York, NY, USA, ESEM '12, pp 49–58

O'Keeffe M, Ó Cinnéide M (2003) A stochastic approach to automated design improvement. In: Proceedings of the 2nd international conference on principles and practice of programming in java. Computer Science Press Inc., pp 59–62

O'Keeffe M, Ó Cinnéide M (2006) Search-based software maintenance. In: Proceedings of the 10th Conference on Software Maintenance and Reengineering, IEEE, Italy, CSMR '06, pp 249–260

O'Keeffe M, Ó Cinnéide M (2008a) Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution 20(5):345–364

O'Keeffe M, Ó Cinnéide M (2008b) Search-based refactoring for software maintenance. Journal of Systems and Software 81(4):502–516

Oliveira Barros de M, Almeida Farzat de F (2013) What can a big program teach us about optimization? In: Proceedings of the 5th International Symposium Search Based Software Engineering, SSBSE'13, pp 275–281

Otero FEB, Johnson CG, Freitas AA, Thompson SJ (2010) Refactoring in automatically generated programs. In: Proceedings of the 2nd International Conference on Search Based Software Engineering, SSBSE'10. Springer, Berlin, Heidelberg, pp 1–2

Ouni A, Kessentini M, Sahraoui H, Hamdi MS (2012) Search-based refactoring: Towards semantics preservation. In: Proceedings of the 28th IEEE International Conference on Software Maintenance, IEEE, ICSM '12, pp 347–356

Ouni A, Kessentini M, Sahraoui H, Boukadoum M (2013a) Maintainability defects detection and correction: a multi-objective approach. Autom Softw Eng 20(1):47–79

Ouni A, Kessentini M, Sahraoui H, Hamdi MS (2013b) The use of development history in software refactoring using a multi-objective evolutionary algorithm. In: Proceedings of the 15th Genetic and Evolutionary Computation Conference, Amsterdam, The Netherlands, GECCO '13, pp 1461–1468

Sahraoui H, Godin R, Miceli T (2000) Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In: Proceedings of the International Conference on Software Maintenance, ICSM '00, pp 154–162

Schäfer M, Thies A, Steimann F, Tip F (2012) A comprehensive approach to naming and accessibility in refactoring java programs. IEEE Trans Softw Eng 38(6):1233–1257

Seng O, Stammel J, Burkhart D (2006) Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the 8th conference on Genetic and Evolutionary Computation, ACM, Seattle, Washington, USA, GECCO '06, pp 1909–1916

Shepperd MJ (1995) Foundations of software measurement. Prentice Hall

Simons C, Singer J, White D (2015) Search-based refactoring: Metrics are not enough. In: Proceedings of the 7th conference on Search-Based Software Engineering, SSBSE '15, vol 9275. Springer International Publishing, pp 47–61

Stuckman J, Wills K, Purtilo J (2013) Evaluating software product metrics with synthetic defect data. In: Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, ESEM '13, pp 259–262

Succi G, Pedrycz W, Djokic S, Zuliani P, Russo B (2005) An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. Empir Softw Eng 10(1):81–104

Tsantalis N, Chatzigeorgiou A (2011) Ranking refactoring suggestions based on historical volatility. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany, CSMR '11, pp 25–34

Ujhazi B, Ferenc R, Poshyvanyk D, Gyimóthy T (2010) New conceptual coupling and cohesion metrics for object-oriented systems. In: Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM'10, pp 33–42

Veerappa V, Harrison R (2013) An empirical validation of coupling metrics using automated refactoring

Weyuker EJ (1988) Evaluating software complexity measures. IEEE Trans Softw Eng 14(9):1357–1365

Williams KP (1998) Evolutionary algorithms for automatic parallelization PhD thesis. University of Reading, UK

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering, 2nd edn. Springer

Mel Ó Cinnéide holds a PhD in Computer Science from Trinity College Dublin, Ireland (2001). He is currently a Lecturer in the School of Computer Science at University College Dublin, Ireland. His main research interests are refactoring, search-based software engineering and software quality. He is a member of the ACM and is a chartered engineer.

Iman Hemati Moghadam holds a PhD in Software Engineering from University College Dublin, Ireland (2014). He is currently assistant professor in the Department of Computer Engineering at Vali-Asr University, Iran. Prior to this, he was a research associate in the Centre for Research on Evolution Search and Testing (CREST) in the Computer Science Department at University College London. His primary research interests are software refactoring, software quality, search-based software engineering and model-driven development.

Mark Harman holds a PhD in Computer Science from the Polytechnic of North London (1992). He is currently Professor of Software Engineering in the Department of Computer Science at University College London, where he directs the CREST centre. His primary research interests are software testing and search based software engineering, a field he co-founded in 2001.

Steve Counsell obtained his PhD in Computer Science from Birkbeck, University of London in 2002. He is currently a Reader in the Department of Computer Science at Brunel University. His research interests relate to empirical software engineering; in particular, refactoring, software metrics and the study of software evolution. He worked as an industrial developer before his PhD and is a Fellow of the British Computer Society.

Laurence Tratt is a Reader in Software Development at King's College London.