On the Link Between Refactoring Activity and Class Cohesion Through the Prism of Two Cohesion-Based Metrics

Steve Counsell[1], Giuseppe Destefanis[1] Steve Swift[1], Mahir Arzoky[1] and Davide Taibi[2]

[1]Department of Computer Science
Brunel University London, United Kingdom
steve.counsell@brunel.ac.uk

[2] CloWeE. Cloud and Web Engineering Research Group
Tampere University, Tampere, Finland
davide.taibi@tuni.fi

***Abstract*—The practice of refactoring has evolved over the past thirty years to become standard developer practice; for almost the same amount of time, proposals for measuring object-oriented cohesion have also been suggested. Yet, we still know very little about their inter-relationship empirically, despite the fact that classes exhibiting low cohesion would be strong candidates for refactoring. In this paper, we use a large set of refactorings to understand the characteristics of two cohesion metrics from a refactoring perspective. Firstly, through the well-known LCOM metric of Chidamber and Kemerer and, secondly, the C3 metric proposed more recently by Marcus et al. Our research question is motivated by the premise that different refactorings will be applied to classes with low cohesion compared with those applied to classes with high cohesion. We used three open-source systems as a basis of our analysis and on data from the lower and upper quartiles of metric data. Results showed that the set of refactoring types across both upper and lower quartiles was broadly the same, although very different in actual numbers. The 'rename method' refactoring stood out from the rest, being applied over three times as often to classes with low cohesion than to classes with high cohesion.**

Keywords— Refactoring, coupling, metrics, empirical.

1. INTRODUCTION

As a core software engineering concept, cohesion has been recognized as an important attribute of software since the late 1960's and is often used in the same context as coupling [1]. In essence, cohesion measures the intra-relatedness of the elements of a component (e.g., the variables within a function, attributes within a class etc). Since the seminal texts on refactoring were published in the 90's by Opdyke [22] and Fowler et al., [13], refactoring has been the subject of hundreds of empirical studies and has become a vital tool in the daily work of a developer [19, 20, 21, 26]. In short, refactoring is the process: "*Of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [13]. In the past twenty-five years or so, a wide range of software metrics have also been proposed that claim to capture OO class features as well as *cohesion* [2, 4]. The most common way of measuring cohesion in the past has been to consider the interplay between instance variables and their usage by the methods of the class. The most well-known of cohesion metrics is the Lack of Cohesion of the Methods in a class (LCOM) of Chidamber and Kemerer (C&K) [6]. A more recent metric by Marcus et al., [16] is the C3 metric which takes a different approach to the measurement of cohesion; it is based on information retrieval techniques and: "*….identifies and captures properties shared between members of a class that take into account not only syntactic but also semantic information*" . Both of these metrics attempt to identify classes where there is low or high class cohesion and, so, in theory, have the same goal. Developers should always strive for high cohesion in their classes, but it is well-understood that as systems age, they tend to decay and class cohesion will tend to deteriorate also [15]. Refactoring is one means through which this decay can be partially reversed and so a direct link exists between the two concepts. Our work is motivated by two factors. Firstly, the concepts of cohesion and refactoring have been around for decades; yet there is no empirical work using systems explicitly linking the two that the authors know of. Both have serious implications for maintenance and test activities. Secondly, the LCOM and C3 metrics both measure cohesion, but in different ways [1]. Our study tries to uncover whether one captures different aspects of code to the other through the prism of refactoring. That may help us understand what each does better (than the other), and also adds to our understanding of refactoring, since it forces us to explore and think about why a developer might take a specific course of action and the consequences if they do not.

In this paper, we therefore explore one over-arching research question. Based on cohesion values generated by the two metrics (i.e., LCOM and C3), we ask whether classes with low cohesion attract different types of refactorings than classes with high cohesion? Developers should spend more time rectifying classes whose elements have become unrelated (i.e., are exhibiting low cohesion) and we might reasonably expect classes with low cohesion to be the subject of different types of refactoring, since their poor structure and composition would dictate other forms of remedial action (i.e., refactoring) compared to highly cohesive classes (in theory). To this end, we used a refactoring dataset produced by Bavota et al. [3], as the basis for our analysis. The same dataset is freely downloadable for replication purposes from their original paper. The data represents information (class-based metrics and the types of refactoring applied) from thousands of refactoring operations applied to releases of three open-source systems. We explore those refactorings with specific reference to cohesion and, to further facilitate our analysis, we decomposed the dataset into inter-quartile ranges. This gave us three parts to our analysis: the lower 25% of cohesion values, the 50% in the mid-range of cohesion values and the upper 25% of cohesion values for each metric. For the purposes of this paper, we ignore the set of mid-range of metric values since we want to explore very high and very low cohesion values. Results showed that, broadly speaking, the same set of refactorings was applied in the upper quarter as in the lower quarter, contrary to what we expected. One major difference between upper and lower quarters related to the Rename field refactoring [13]. Well over three times as many of these refactorings were applied to classes with low cohesion as to classes with high cohesion. This implies that there might be strong links between the cohesion of a class and the ability of a developer to understand firstly, what the class actually does and secondly, how refactoring can be used to remedy that situation. The remainder of the paper is organized as follows. In the next section, we describe preliminary information on the two metrics we use for our analysis, the three systems studied, the data collected and summary data. We then present results through an analysis of each of the three systems (Sections 3 and 4) using the LCOM and C3 metrics. Section 5 discusses related work and general discussion points as well as threats to study validity. Finally, we conclude and point to further work (Section 6).

2. PRELIMINARIES

2.1 The LCOM metric

In this paper, we follow the definition of LCOM according to the JHawk tool [31]; this tool was used to collect all the metrics used in this paper. Informally, the LCOM metric measures cohesion through the distribution of class instance variables across the methods of a class. If every method uses every instance variable, then that class is maximally cohesive. If, at the other extreme, the instance variables used by every method are completely disjoint, then the class has minimal cohesion. Consider, for example, a class C with three methods we will call M1, M2 and M3 and a set of instance variables {a, b, c, d, e, x, y, z}. Let I1-I3 be the three sets of instance variables used by each of the three methods where {I1} = {a, b, c, d, e}, {I2} = {a, b, e} and {I3} = {x, y, z}. LCOM is the number of empty intersections of I1-I3, minus the number of non-empty intersections (the calculated value is set to zero if the subtraction is negative). Enumerated: {I1} ∩ {I2} is non-empty, but {I1} ∩ {I3} and {I2} ∩ {I3} are empty sets. LCOM, in this case, is therefore 2 (number of empty sets) - 1 (non-empty set) giving a value for LCOM of 1. It then follows that the larger the value of LCOM, the less cohesive the class and the lower the LCOM, the more cohesive the class.

2.2 The C3 metric

The C3 metric is calculated using information recorded in the source code through information in the methods. Analysis of this type of *semantic information* can be useful in forming and evaluating metrics. The code being analyzed is converted into a text corpus and only identifiers and comments are extracted from each method. According to Marcus et al., [16] the process is as follows: "*Each method is a document in this corpus and LSI* […Latent Semantic Indexing [9]] *is used to map each document to a vector in a multidimensional space determined by the terms that occur in the vocabulary of the software. Once each method is represented as a vector, a similarity measure between any two methods can be defined as the cosine between their corresponding vectors. This similarity measure will express how much relevant semantic information is shared among the two methods, in the context of the entire system*". This is the basis upon which the C3 metric is calculated. A graph-based representation of the system with weighted edges is used to compute the similarity of pairs of methods. If a class is cohesive, then C3 will tend towards a value of 1, meaning that all the methods of the class are strongly related to each other conceptually. If the methods are weakly related conceptually however, then the C3 value will tend towards a value of 0 and is considered to have low cohesion.

2.3 Systems studied

The systems studied consisted of three Java open source projects: Xerces [30], ApacheAnt [28] and ArgoUML [29]. Xerces-J is a Java XML parser, Apache Ant a build tool and library primarily designed for Java applications and ArgoUML a UML modeling tool and. Table 1 is taken verbatim from [3] and shows the salient characteristics of the three systems. Here, 'Rel.' is the number of releases and the final column (#Ref.) represents the total number of refactorings for that system before we decomposed it into its inter-quartile ranges.

Table 1. Features of the three systems analyzed from [3]

System	Period	Analyzed	Rel.	# Classes	# Ref.
Xerces	Nov '99-Nov '10	1.0.4-2.9.1	33	181-776	7502
Apache Ant	Jan '00-Dec '10	1.2-1.8.2	17	87-1191	1289
ArgoUML	Oct '02-Dec '11	0.12-0.34	13	777-1519	3255

Table 2 shows data the number of refactorings in each of the ranges for the three systems, after it had been decomposed. It shows the Upper Quartile (UQ) and Lower Quartile (LQ) median values for the LCOM and C3 metrics and the Inter-Quartile Range (IQR) in each case. IQR is calculated as the UQ value minus the LQ value. For example, in the UQ of the Xerces systems, the median LCOM was 5 and in the lower quartile the median was 1724. The IQR for LCOM was 1719. For the C3 metric UQ, the median C3 value was 0.36 and was 0.21 for the LQ. The IQR was 0.21.

Table 2. Dataset decomposition into quartiles

System	**LCOM (UQ)**	**LCOM (LQ)**	**IQR**	**C3 (UQ)**	**C3 (LQ)**	**IQR**
Xerces	1724	5	1719	0.36	0.15	0.21
ApacheAnt	388	2	386	0.35	0.12	0.23
ArgoUML	164	0	164	0.51	0.14	0.37

The Ref-Finder tool [14] was used to extract the set of refactorings on which our analysis was based; the refactorings were extracted and validated as part of the earlier study and are reused in our paper [3]. The Ref-Finder tool collects up to sixty-three of Fowler's original set of 72 refactorings [13] and has recall of 95% and precision of 79%. We begin our analysis by looking at the LCOM metric and the refactorings in the UQ and LQ quartiles, before moving on to do the same for C3. At this juncture, we emphasize the point that a high numerical value of LCOM means that the class has low cohesion, since the metric measures the 'lack' of cohesion in a class. Correspondingly, a low value of LCOM implies that the class is relatively cohesive (i.e., it does not lack cohesion). It is the opposite of C3 where a high cohesion tends towards 1 and low cohesion 0.

3. LCOM REFACTORING ANALYSIS

Table 3 shows, for the Xerces system, the number of refactorings applied in the UQ and LQ when ranked on the five most popular refactorings in the UQ. For example, the most popular refactoring in the UQ of the LCOM values (i.e., where classes have low cohesion) was the Rename method refactoring (RM) with 573 occurrences. This represents 26.91% of the total number of refactorings in the UQ. In the LQ, where classes have high cohesion, the corresponding number of RM refactorings was just 110, representing just 6.63% of the total number of refactorings in that quartile. A totals row in the table shows the sum of numbers in each of the columns. Below the totals row, we also show, for completeness, and for each system, the refactorings that were in the top five refactorings in the LQ. So, for example, from Table 3, the top five refactoring in the UQ are in the row order we see (i.e., RM, MM, MF, AP and CDCF); for the LQ, the top five most popular refactorings were AP, RP, RMN, MF and MM, in that order. The motivation for RM is when the name of a method does not convey what that method does very well; it should therefore be renamed to make its purpose more obvious/clear. We note that our choice of the top five refactorings was made on the basis that this gave us the majority of what we believe to be key refactorings for that system. One plausible explanation for the large number of RM refactorings in the UQ could be down to poor naming of methods and a general lack of understandability about what the class does by developers; a class with low cohesion will be difficult to understand. If developers cannot understand what the class functionality is or what the methods in the class do, then this might naturally be the trigger for the rename method refactoring to be applied.

Table 3. Refactoring data for the Xerces system

Refactoring	**LCOM (UQ)**	**%**	**LCOM (LQ)**	**%**
Rename Method (RM)	573	26.91	110	6.63
Move Method (MM)	250	11.74	121	7.30
Move Field (MF)	244	11.46	199	12.00
Add Parameter (AP)	182	8.55	325	19.60
Cons. Dupl. Frag. (CDCF)	131	6.15	96	5.79
Totals	**1380**	**64.81**	**851**	**51.32**
Remove Parameter (RP)	100	4.70	223	13.45
Replace Magic Number (RMN)	109	5.12	222	13.39

It is remarkable that the largest proportion of refactorings in the UQ was for RM, but for the LQ (where we consider classes to have relatively high cohesion) it was the Add Parameter (AP) refactoring that dominated. The motivation for using AP is when [13]: "*A method doesn't have enough data to perform certain actions*". The solution is to: "*Create a new parameter to pass the necessary data*". One possible explanation for the result for AP is that adding a parameter means that data available to the class is being shared around the methods of the class. This in theory contributes positively to the cohesiveness of a class. After all, the basis of the LCOM metric is the commonality in the use of instance variables by the methods; if greater sharing is taking place (through the addition of parameters) then that can only aid its cohesiveness. Table 3 also shows significant numbers of Move Field (MF) and Move Method (MM) refactorings in both the UQ and LQ. The motivation for applying the MM refactoring is when [13]: "*A method is used more in another class than in its own class*". The solution is to: "*Create a new method in the class that uses the method the most, then move code from the old method to there*." A similar motivation and solution applies to the MF refactoring. One generally recognized and accepted way of improving cohesion is to move features around classes to where they are most needed. This is normally a convenient way of reducing coupling also; however, in a positive sense, reducing coupling in one or more classes can often lead to improved class cohesion in other classes if external features are moved around. So, the motivation for applying MF and MM might be to reduce coupling, but this has the side effect of improving cohesion. Finally, the Consolidate Duplicate Conditional Fragments (CDCF) refactoring was applied in similar numbers in the UQ and LQ. The CDCF refactoring removes duplicate lines of code in conditional statements. Table 4 shows the same data for LCOM as that in Table 3, but for the ApacheAnt system.

Table 4. Refactoring data for the ApacheAnt system

Refactoring	**LCOM (UQ)**	**%**	**LCOM (LQ)**	**%**
Rename Method (RM)	90	23.75	6	1.82
Inline Temp (IT)	37	9.76	6	1.58
Introduce Exp. Var (IEV)	36	9.50	24	7.27
Remove Parameter (RP)	33	8.71	36	10.91
Add Parameter (AP)	30	7.92	43	13.03
Total	**226**	**59.64**	**115**	**34.61**
Replace Magic No. (RMN)	32	8.44	113	34.24
Cons. Dup. Frag. (CDCF)	11	2.90	25	7.58

The RM refactoring is, again, the most frequently applied refactoring in the UQ for the ApacheAnt system (90 compared with just 6 in the LQ) and the AP again features, as it did for the Xerces system. The numbers of AP refactorings in the LQ far exceed those in the UQ - the same result as was found for AP in the Xerces system. The number of RP refactorings in the LQ for the Xerces system was 223, compared with just 100 in the UQ. The motivation for the RP refactoring is when a parameter in the body of a method is unused. The RP refactoring simply removes that parameter. The most frequently applied refactoring across both the UQ and LQ was Replace Magic Number (the full title of this refactoring [13] is 'Replace Magic Number with Symbolic Constant'). The motivation for this refactoring is when a value is hard-coded in multiple places throughout the code. It should be replaced with a `const` declaration for the hard-coded value. The example of "pre-" refactoring and "post-" refactoring code for RMN given in [13] is as follows:

Before refactoring:

```
function potentialEnergy(mass, height) {
  return mass * 9.81 * height;
}
```

becomes:

```
const STANDARD_GRAVITY = 9.81;
function potentialEnergy(mass, height) {
  return mass * STANDARD_GRAVITY * height;
}
```

Table 5 shows the corresponding data for the ArgoUML system. The AP and RP refactoring again feature strongly in the UQ data; however, one of the largest disparities between UQ and LQ in percentage terms is for the RM refactoring, again dominating in terms of numbers in the UQ. Exactly the same observation was made for the previous two systems in terms of RM and so a pattern is emerging for this particular refactoring. The MF refactoring dominated the LQ; 27.91% of refactorings in the LQ were MF.

Table 5. Refactoring data for the ArgoUML system

Refactoring	**LCOM (UQ)**	**%**	**LCOM (LQ)**	**%**
Add Parameter (AP)	144	15.37	53	11.65
Remove Parameter (RP)	134	14.30	59	12.97
Rename Method (RM)	110	11.74	16	3.52
Remove Control Flag (RCF)	98	10.46	10	2.20
Move Method (MM)	82	8.75	61	13.41
Totals	**568**	**60.62**	**199**	**33.75**
Move Field (MF)	30	3.20	127	27.91
Introduce Expl. Var. (IEV)	32	3.42	62	13.63
Replace Meth with Obj.	75	8.00	60	13.19

It is clear from the analysis thus far that the standout refactoring is RM and, to a lesser extent, AP and RP. It is also revealing that for the ApacheAnt and ArgoUml systems, we found no occurrences of inheritance-related refactorings such as Pull up field, Pull up method, Push down method/field. In the Xerces system, we only found limited evidence (129 out of 7503) of these refactorings. In none of the three systems did we find evidence either that classes had been decomposed using the Extract class refactoring [13]; according to Du Bois et al., [11] this is one of the key refactorings that aid the cohesion of class.

3.1 Does class size matter?

Across all three systems and considering the LCOM metric overall, the RM refactoring was consistently found to be the most frequently applied refactoring, the vast majority of which were applied to classes in the UQ in the case of the LCOM (i.e., to those classes with a high LCOM and hence low cohesion) and the LQ in the case of the C3 metric (low cohesion classes). This suggests that these classes may have poor readability and hence comprehension issues, either intrinsically or as a result of constant change to the class requiring the method name changes to be applied. Classes with high cohesion, on the other hand, were not targeted as much by the RM refactoring. The AP and RP refactorings also figure across all three systems, but the pattern is a little less clear in these two cases in the UQ and LQ.

One factor that may have influenced this result for RM is the size of classes where it was applied. Larger classes are more difficult to understand (generally speaking) and if classes are larger, then they will inevitably attract more RM than smaller classes by the law of averages. To investigate this question, we therefore explored whether classes in the UQ were larger than those in the LQ for the LCOM and *vice versa* for the LCOM metric. Figure 1 for the Xerces system shows the plot of class size (*y*-axis) given by the number of methods for each refactoring (on the *x*-axis); the UQ (thicker line) and LQ (thinner line) represent each LCOM observation on the UQ and LQ, respectively. The LQ values appear to exceed those in the LQ from the figure. In fact, the average number of methods for the UQ was 100.99 (median 58) and for the LQ the average was 160.99 (median 148). In other words, the size of classes *was* generally lower in the UQ where

LCOM was higher and hence cohesion lower. Excessive differences class sizes between the UQ and LQ would not seem to be the immediate reason for the mismatch in renaming activity.

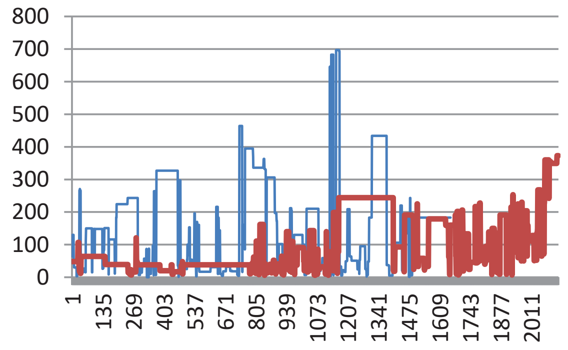

Figure 1. No. methods for the Xerces system (UQ vs. LQ)

For the ApacheAnt system, a similar trend was observed as that in Figure 1. The average number of methods for the UQ was 82.86 (median 44) and for the LQ the average was 106.10 (median 121). In other words, the size of classes was again generally lower in the UQ where LCOM was higher and hence cohesion lower. Finally, for the ArgoUML system, the average number of methods in the LQ was 9.00 (median 4) and for the UQ, average 35.78 (median 19). For two of the systems therefore, the number of methods in the LQ was much higher than for the UQ.

4. C3 REFACTORING ANALYSIS

Just as we analyzed the LCOM metric in the previous section, we next compare the results for the C3 metric. Since LCOM and C3 compute cohesion in very contrasting ways, we might expect to see different results from our analysis in terms of what refactorings are applied to classes with high and low cohesion. The C3 metric produces a different range of values (all lay between 0-1) and the C3 metric does not use instance variables as a basis of its computation. Table 6 shows the C3 data for the Xerces system (same format as produced for LCOM). The data mirrors that in Table 3, but is clearly very different in terms of the most popular five refactorings.

Table 6. Refactoring data for the Xerces system

Refactoring	**C3-UQ**	**%**	**C3-LQ**	**%**
Add Parameter (AP)	459	21.56	162	9.90
Remove Parameter (RP)	236	11.09	108	6.60
Move Method (MM)	233	10.94	209	12.78
Rename Method (RM)	230	10.80	471	28.79
Replace Magic Number (RMN)	198	9.07	132	8.07
Totals	**1356**	**63.46**	**1082**	**66.14**
Move Field (MF)	138	6.48	193	11.80

The most marked difference between the UQ and LQ data is again for the RM refactoring. Nearly 29% of the top five refactorings in the LQ were accounted for by RM (numbering 471), the largest of any percentage in the data thus far, whether in the UQ or LQ. A low value of C3 represents low cohesion and so we see exactly the same result occurring as was found in Tables 3-5 for the LCOM metric; namely, that classes with low cohesion attract large numbers of RM refactorings. Clearly, there is something that links classes with low cohesion and the propensity for applying the RM refactoring and this may have important implications for maintainability – perhaps proper naming of methods, one of the clean code principles impacts class erosion more than we think it does [18]. It is also interesting that the AP and RP refactorings are again the most prevalent in the UQ of the C3 data representing highly cohesive classes and reflecting the result that we also found for LCOM. Table 7 shows the data for the ApacheAnt system. The most popular refactoring in the UQ was for the RMN refactoring (accounting for nearly 40% of refactorings). While the RM does not figure in the top five refactorings in the UQ, it does appear to be the most popular refactoring in the LQ data, by a considerable margin. Nearly 35% of all refactorings in the LQ were RM refactorings.

Table 7. Refactoring data for the ApacheAnt system

Refactoring	**C3-UQ**	**%**	**C3-LQ**	**%**
Replace Magic Number (RMN)	145	38.29	15	4.55
Remove Assign. to Par. (RAP)	35	9.23	0	0.00
Cons. Dup. Frag. (CDCF)	30	7.92	0	0.00
Introduce Expl. Variable (IEV)	28	7.39	39	11.82
Add Parameter (AP)	26	6.86	39	11.82
Totals	**264**	**69.69**	**93**	**28.19**
Rename Method (RM)	9	2.37	114	34.55
Move Field (MF)	6	1.58	43	13.03
Remove Parameter (RP)	22	5.80	39	11.82

The result for the RMN refactoring in Table 7 is also interesting. One of the advantages of applying this metric is that it improves maintainability, since you only have to change a constant's value once if it is declared and set in one place. The majority of the RMN refactorings were in the UQ, i.e., for highly cohesive classes. One possibility why so many of these refactorings were applied was because declaring a `const` and then using it many of the methods of a class adds to the textual similarity between methods. In other words, the C3 metric would see application of this refactoring as something that contributed positively to the cohesion of a class; the LCOM metric would not, since it uses instance variables only and is not based on patterns of similarity in other parts of the class. Finally, Table 8 shows the data for the ArgoUML system. Nearly a third of refactoring (29.78%) were accounted for by the MF refactoring. Again, we need to look to the way that the C3 metric is calculated to provide an explanation for this result. Moving a field to where it is used more from outside immediately raises the textual similarity within the class, since, by definition, you would only move a field to a class where it is being used most. If many elements of the class use that data, then that raises the textual similarity of the elements in the class and hence the value of the C3 metric. It is another case of where reducing coupling would have the positive side-effect of raising cohesion in the target class. The RM is the second highest of refactorings in the LQ and did not feature in the top five refactorings of the UQ. Only Replace Method with Method Object was applied more times. The motivation for using the RMwMO refactoring is when [13]: "*You have a long method in which the local variables are so intertwined that you can't apply Extract Method*". The mechanics of this refactoring are: "*Transform the method into a separate class so that the local variables become fields of the class. Then you can split the method into several methods within the same class*".

Table 8. Refactoring data for the ArgoUML system

Refactoring	**C3-UQ**	**%**	**C3-LQ**	**%**
Move Field (MF)	279	29.78	46	10.11
Move Method (MM)	143	15.26	52	11.43
Add Parameter (AP)	123	13.13	38	8.35
Remove Parameter (RP)	111	11.85	45	9.89
Replace Method with Obj.	90	9.61	59	12.97
Totals	**746**	**69.63**	**240**	**52.75**
Rename Method (RM)	32	3.42	56	12.31

Clearly, the C3 metric showed similar results to that of LCOM.

4.1 Does class size matter?

The same result was found for the C3 metric as for the LCOM with respect to RM - significantly more were applied to classes with low cohesion than with high cohesion. The same question has to be asked as to whether the size of the class in terms of number of methods may have been a factor in attracting that number of RM refactorings. Looking at the C3 values for Xerces, the average number of methods for the UQ was 67.16 (median 47) and for the LQ the average was 227.11 (median 244). In other words, the size of classes *was* generally lower in the UQ where C3 was higher and hence cohesion higher. This contradicts the view that the size of the classes in the system may have been the reasons for the large number of RM refactorings. For the ApacheAnt system, the average number of methods for the UQ was in 39.99 (median 23) and for the LQ the average was 117.49 (median 121), again contradicting the view about large classes. Finally, for the ArgoUML system, the UQ the average number of methods was 31.96 and median 16; for the LQ the average was 32.52 and median 25. For the ArgoUML system, the method sizes were comparable. Again, we can say that the pattern in number of methods from a summary analysis does not suggest that it was a factor in the RM refactoring trend.

As a final aspect of the analysis, it is worth considering the (correlational) relationship between the LCOM and C3 for the UQ and LQ set of values, since that might inform our understanding of the results. Table 9 shows the Spearman (Sp.), Kendall (Kn.) and Pearson (Ps.) correlation coefficients for LCOM versus C3 in the UQ and LQ (in that order). Spearman and Kendall are both non-parametric measures making no assumption about the distribution of the data; Pearson's measure however assumes a distribution in the data (usually normal). We include both all types for completeness. Here an '*' indicates statistical significance at the 0.01 level; we note that in the LQ of ArgoUML, all LCOM values were zero, hence correlation coefficients could not be computed.

Table 9. Correlation values between LCOM and C3

System	**UQ (Sp., Kn., Ps.)**			**LQ (Sp., Kn., Ps.)**		
Xerces	-0.62*	-0.44*	-0.50*	-0.31*	-0.22*	-0.29*
ApacheAnt	0.56*	0.31*	0.22*	-0.37*	-0.27*	-0.32*
ArgoUML	0.16*	0.11*	0.03	-	-	-

The most striking aspect of Table 9 is the difference in the UQ between the three systems. For the Xerces system, the correlation values are highly significant (negatively), while for ArgoUML they are significant (positively). Reasonably, we may expect a negative correlation between the two metrics, since one measures the strength of cohesion and the other the lack of cohesion. The explanation for this data is relatively straightforward: values of the LCOM metric increase exponentially with the number of attributes, while the range of the C3 metric is 0-1 and where lower values represent low cohesion (the opposite to LCOM). The correlations demonstrate that the Xerces system may have a different, yet more "conformant" set of classes when compared to that of ApacheAnt and ArgoUML, at least for the UQ.

5. RELATED WORK/DISCUSSION

Cohesion has been the subject of numerous empirical studies, yet remains, in many ways, one of the most controversial topics in the software metrics community. Although cohesion as a concept was first introduced in the 1970's and for the procedural paradigm [25], these days it is more often than not seen as part a study of the OO paradigm. The gold standard of cohesion metrics is still the LCOM, even though numerous attempts have been made to study and/or improve on it in the past twenty-five years [5, 12, 24]. This includes revisions of the original metric itself and claims by other metrics that they are an improvement. The reasons why it has persisted is manifold; however, the two key reasons that we believe it persists are that most OO data collection tools collect the C&K metrics as standard and those tools have enjoyed and will continue to enjoy widespread use; secondly, because of the large number of previous studies that have used LCOM, it is difficult to justify using any other cohesion metrics without a sound empirical grounding showing demonstrably that it is better or more useful in some sense. That in itself is problematic since the LCOM has been used for over twenty-five years. What we have tried to do in the paper presented is to highlight the links between two cohesion metrics and refactoring. We make no judgment on either metric as to which is better since they are both founded on reasonable assumptions and theory. The original paper in which the C3 metric paper was first published was later applied to defect-proneness [17]. In that later paper, it was shown how the C3 improved upon current measures of cohesion. The study also showed how combining C3 with existing structural cohesion metrics was a better predictor of defective classes when compared to different combinations of structural cohesion metrics. The problem with adopting the metric is that very few other studies have used the metric.

The work in our paper was inspired by previous work of the same authors [8], where coupling levels in the same three systems were analyzed. The Coupling between Objects (CBO) metric of C&K and the Conceptual Coupling between Classes (CCBC) metric of [23] were compared from a refactoring perspective. The thrust of the research was whether coupling levels given by the two metrics were influenced by specific refactoring effort. Results showed no significant difference in the types of refactoring applied across either coupling for both metrics; refactorings usually associated with coupling removal were actually *more* numerous for classes with low levels of coupling in some cases. A lack of inheritance-related refactorings across all systems was also noted. The over-riding message was that developers were largely indifferent to classes with high coupling when it came to refactoring types – they treat classes with relatively low coupling in almost the same way. It is interesting in this paper that through the C3 metric especially, cohesion can improve if class features are moved around since they add to the textual and hence semantic similarity of methods. Another relevant and interesting study on refactoring is by Du Bois et al., [11]. The paper explored different types of refactoring and their relevance in the improvement of cohesion and coupling in systems. Rationale for using a subset of refactorings was provided in the context of why and why they would help improve cohesion and/or coupling. Results from their analysis showed that for cohesion *and* coupling, the best refactoring was the Move method refactoring [27]. "Good" results were provided by the guidelines on, amongst others, the Replace method with method object and Extract class refactorings. For the last two refactorings, the authors point out that "*These guidelines are a good help in improving cohesion, yet provide only limited help in resolving coupling issues*". On the down side, the paper did not use any empirical system data for the analysis and did not consider a wide range of refactorings. The RM refactoring was excluded from their analysis, which, on the evidence of the work presented, seems to be inextricably linked to low cohesion and we found no examples of the Extract class refactoring across the data we used. In the paper presented herein, we implicitly provide a comparison of the LCOM and C3 metrics. Briand et al., present a comprehensive appraisal and comparison of the different forms of the LCOM metric and interpretations from an empirical and theoretical standpoint [5]. In a similar way, Counsell et al. [7] also explored three cohesion metrics from a theoretical and empirical standpoint. One of their conclusion resonates with the work we present: "…*While it may be true that a generally accepted formal and informal definition of cohesion continues to elude the OO software engineering community, there seems considerable value in being able to compare, contrast, and interpret metrics which attempt to measure the same features of softwar*e".

5.1 Discussion

The preceding analysis raises a number of questions about the results. Firstly, as found in Bavota et al., [3], and supported in this paper, developers only seem to use a limited number of refactorings extensively. To put this into perspective, from our analysis of the five most popular refactorings in the upper and lower quartiles, only thirteen distinct refactorings can be identified from the sixty-three that Ref-Finder is capable of extracting. It may be true that there are many other refactorings embedded in the code that are not part of the tool and exploring this aspect of the data is a topic of future work. Conspicuous by their absence are any inheritance-related refactorings in the top five of either the upper or lower quartiles, a trend also identified in the previous coupling analysis [8]. Secondly, we have to consider what the analysis means for the two metrics studied. Clearly, both metrics are connected to a common core of refactorings. If we consider the set of thirteen refactorings across all Tables 3-8, there is only one refactoring, i.e., Remove Assignment to Parameters in Table 7 that is not common to either of the LCOM and C3 metrics top five (upper or lower quarter). The question, again a topic for future work, is to establish whether the two metrics essentially measure the same thing. Our analysis has not shown any significant or obvious differences, although for some refactorings that we've highlighted the approach that the C3 metric takes with respect to method textual similarity would likely make a difference to its value compared with LCOM. Thirdly, we need to consider the implications of the results of our analysis on the developer. It is clear that the Rename method refactoring is different in terms of how often it is applied compared with other refactorings – this is reasonably clear. We have to consider the possibility that it is the simplicity of this refactoring that makes it an easy target for developers and the fact that many tools support the refactoring; the same refactoring also poses a low test burden [10] compared with other refactorings and this may also make it an appealing refactoring if it is being applied manually.

Our study also needs to consider the threats to the validity of the work presented. Firstly, we have only considered three open-source systems and this poses a validity threat since it brings into question the generalizability of the work. Secondly, we have only considered the top five refactorings in our analysis; while we could have considered every refactoring, this would have diluted the message of the paper and made the analysis excessively cumbersome. It would also have made the message that were trying to convey less focused. A third threat is that we have not included information about the *effect* that each refactoring has on the class where it is applied. Put another way, we know that, for example, a method has been renamed, but we don't know whether that refactoring had a positive effect or not on the cohesion of the class. We have assumed that refactorings have been applied in a remedial sense. In the case of Rename method we have posited that it was done to make the method and class more readable and hence more maintainable. Perhaps this was the first step in tidying up classes and making them more cohesive in the long run (again in accordance with the notion of clean code [18]. Finally, without experimenting with actual developers to find out how their decision-making processes work and on a scale that can be generalized, our conclusions will always be speculative.

6. CONCLUSIONS AND FURTHER WORK

In this paper, we analyzed large set of refactorings to understand the characteristics of two cohesion metrics (the LCOM [6] and the C3 [16]) from a refactoring perspective. Our main research question was whether, when we decomposed the metric data into quartiles, different sets of refactorings would be identified in the lower and upper quartiles for each metric. Results showed that the set of refactoring types across both upper and lower quartiles was broadly the same, although in actual numbers the refactorings differed widely. The Rename method refactoring stood out from the rest for both metrics. It was applied over three times as often to classes with low cohesion than classes with high cohesion. This suggests that it may be a useful tool in making classes more cohesive for long-term maintainability, although a more in-depth study would need to establish how that would work in practice. We believe that the study is interesting since it tells us something about the relationship between the concept of cohesion and refactoring, an under-researched, yet important aspect of systems. There are many other avenues for further work. Firstly, it would be useful to include defect data in a similar analysis to establish whether low or highly cohesive classes as we have portrayed them influence the type of refactorings applied to a class. It is also planned to undertake experiments with industrial developers to determine the motivation for why developers refactor and when. It would also be interesting to measure the post-impact of renaming refactorings on the comprehension of classes, since our study suggests that it may be a significant factor. Finally, we want to expand the analysis to other open-source systems and to include industrial systems to see how the two compare.

REFERENCES

[1] R. Barker, E. Tempero, (2008). A Large-Scale Empirical Comparison of Object-Oriented Cohesion Metrics. APSEC, 414-421.

[2] V. Basili, L. Briand, W. Melo, 1995. A validation of object-oriented design metrics as quality indicators. IEEE Trans. on Software Eng. 22, 10, 751–761.

[3] G. Bavota, A. De Lucia, M. Di Penta,, R. Oliveto, F. Palomba, An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Software. 107, 1-14.

[4] L.C. Briand, J. Daly, J. Wüst. 1999, A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Trans. Softw. Eng. 25, 1, 91-121.

[5] L. Briand, J. Daly and J. Wust, A Unified Framework for Cohesion Measurement in Object-Oriented Systems, Empirical Software Engineering, Vol. 3, No. 1, 1998, pp. 65-117.

[6] S. R. Chidamber, C. F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, 20(6):476-493, 1994.

[7] S. Counsell, S. Swift, J. Crampton, The interpretation and utility of three cohesion metrics for object-oriented design. ACM Transactions on Softw. Eng. Methodology. 15, 2 (April 2006), 123-149.

[8] S. Counsell, M. Arzoky, G. Destefanis, D. Taibi, On the Relationship Between Coupling and Refactoring: An Empirical Viewpoint. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Porto De Galinhas, Brazil, pages 1-6, 2019.

[9] S. Deerwester, S., Dumais, G, Furnas, T., Landauer, and R. Harshman, Indexing by Latent Semantic Analysis, Journal of the American Society for Information Science, vol. 41, 1990, pp. 391-407.

[10] A. van Deursen and L. Moonen. The Video Store Revisited - Thoughts on Refactoring and Testing. International Conf. on eXtreme Programming and Flexible Processes in Software Engineering XP 2002, Sardinia, Italy.

[11] B. Du Bois, S. Demeyer, J. Verelst, Refactoring -- Improving Coupling and Cohesion of Existing Code. WCRE 2004: 144-151.

[12] L. Etzkorn, S. Gholston, J. Fortune, C. Stein, D. Utley, P. Farrington and G. Cox, A comparison of cohesion metrics for OO systems, Information and Software Technology, vol. 46, no. 10, August 2004, pp. 677-687.

[13] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[14] M. Kim, M. Gee, A. Loh, N. Rachatasumrit, Napol, (2010). Ref-Finder: A refactoring reconstruction tool based on logic query templates. ACM SIGSOFT Symposium on the Foundations of Software Eng.. 371-372.

[15] M. Lehman, (1980), On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. Journal of Systems and Soft., 1: 213–221.

[16] A. Marcus, Denys Poshyvanyk: The Conceptual Cohesion of Classes. International Conference on Software Maintenance, ICSM 2005: 133-142

[17] A. Marcus, D. Poshyvanyk, R. Ferenc, Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. IEEE Trans. Software Eng. 34(2): 287-300 (2008).

[18] R. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008

[19] T. Mens, T. Tourwe, 2004. A survey of software refactoring. IEEE Transactions on Software Engineering 30, 2, 126–139.

[20] E. Murphy-Hill, C. Parnin, A. Black, How We Refactor, and How We Know It. IEEE Trans. Software Eng. 38(1): 5-18 (2012).

[21] S. Negara, N. Chen, M. Vakilian, R. Johnson, D. Dig, A Comparative Study of Manual and Automated Refactorings, ECOOP 2013.

[22] W. Opdyke, Refactoring object-oriented frameworks, PhD Thesis, University of Illinois, Urbana-Champaign, 1992.

[23] D. Poshyvanyk, A., Marcus, R., Ferenc,, and T., Gyimothy, 2009. Using information retrieval based coupling measures for impact analysis. Empirical Software Engineering 14, 1, 5–32.

[24] C. Stein, G. Cox, L. Etzkorn, Exploring the relationship between cohesion and complexity, Journal of Computer Science. 1 (2): 137–144, 2005.

[25] W. Stevens, G. Myers, L. Constantine, Structured design, IBM Systems Journal. 13 (2): 115–13, 1974.

[26] N. Tsantalis, A., Chatzigeorgiou, A. 2009. Identification of move method refactoring opportunities. IEEE Trans. on Soft. Engineering 35, 3, 347–367.

[27] https://refactoring.guru/move-method

[28] http://ant.apache.org/

[29] http://argouml.tigris.org/

[30] http://xerces.apache.org/xerces-j/

[31] http://www.virtualmachinery.com/