Cohesion as Changeability Indicator in Object-Oriented Systems

Hind Kabaili, Rudolf K. Keller and François Lustman
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
E-mail: {kabaili / keller / lustman}@iro.umontreal.ca

Abstract

The assessment of the changeability of software systems is of major concern for buyers of large systems found in fast-moving domains such as telecommunications. One way of approaching this problem is to investigate the dependency between the changeability of the software and its design, with the goal of finding design properties that can be used as changeability indicators. In the realm of object-oriented systems, experiments have been conducted showing that coupling between classes is such an indicator. However, class cohesion has not been quantitatively studied in respect to changeability. In this research, we set out to investigate whether cohesion is correlated with changeability. As cohesion metrics, LCC and LCOM were adopted, and for measuring changeability, a change impact model was used. The data collected on three test systems of industrial size indicate no such correlation. Manual investigation of classes supposed to be weakly cohesive showed that the metrics used do not capture all the facets of class cohesion. We conclude that cohesion metrics such as LCC and LCOM should not be used as changeability indicators.

Keywords: Software quality, cohesion, changeability, change impact, correlation, software metrics, C++ language.

1. Introduction

The use of object-oriented (OO) technology for developing software has become quite widespread. Researchers assert that OO practice assures good quality software, that is, particularly software that is easy to maintain, reuse, and extend. Industrial buyers of OO software want to be sure of the product quality they acquire. For this, they need OO measures, to evaluate the software they consider buying.

For various reasons, Bell Canada, the industrial partner in this project, is interested in buying large-scale software rather than developing it. Finding practical ways to assess the quality of software is an important element in the software purchasing approach of the company. By practical, we mean automated and easy to implement.

The *SPOOL* project *(Spreading desirable Properties into the design of Object-Oriented, Large-scale software systems)* is a joint industry/university research project between the *Quality Engineering and Research* team of *Bell Canada* and the *GELO* group at the *Université de Montréal*. As part of the project, design properties are investigated as changeability indicators.

Cohesion is an important property of OO designs, and metrics have been proposed to quantify and measure it. In this paper, we try to assess cohesion as an indicator of changeability. The paper is organized as follows. Section 2 presents an overview of cohesion as a quality indicator and describes the change impact model used in the experimentation. The relationship between cohesion and changeability was tested empirically, as reported in Section 3. The negative result of the test led us to investigate the reasons behind this lack of relationship as described in Section 4. Section 5, finally, summarizes the work and provides an outlook into future work.

2. Cohesion and changeability

Building quality OO systems relies on good design. To assess the quality of a design with some objectivity, we need to quantify design properties. Several software metrics have been developed to assess and control the design phase and its products [1,6,7,9]. One of the most important design properties is cohesion. Module cohesion was introduced by Yourdon and Constantine as "how tightly bound or related the internal elements of a module are to one another" [15]. A module has a strong cohesion if it represents exactly one task of the problem domain, and all its elements contribute to this single task. Yourdon and

Constantine describe cohesion as an attribute of design, rather than code, and as an attribute that can be used to predict reusability and maintainability. However, these assumptions have never been supported by experimentation.

2.1. Cohesion in object-oriented systems

A class is cohesive if it cannot be partitioned into two or more sets defined as follows. Each set contains instance variables and methods. Methods of one set do not access directly or indirectly variables of another set. Many authors have implicitly defined class cohesion by defining cohesion metrics. In the OO paradigm, most of the cohesion metrics are inspired from the metric suite defined by Chidamber and Kemerer (C&K) [6]. According to these authors "if an object class has different methods performing different operations on the same set of instance variables, the class is cohesive". As a metric for assessing cohesion, they define LCOM (Lack of Cohesion in Methods) as the number of pairs of methods in a class, having no common attributes, minus the number of pairs of methods sharing at least one attribute. The metric is set to zero when the value is negative.

Li and Henry [11] redefine LCOM as the number of disjoint sets of methods. Each set contains only methods that share at least one instance variable.

Hitz and Montazeri [9] restate Li's definition of LCOM based on graph theory. LCOM is defined as the number of connected components of a graph. Vertices represent methods. There is an edge between 2 vertices if the corresponding methods access the same instance variable. Hitz and Montazeri propose to split a class into smaller, more cohesive classes, if LCOM > 1.

Bieman and Kang [2] propose TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) as cohesion metrics, based on Chidamber and Kemerer's approach. They too consider pairs of methods using common instance variables. However, the way in which they define attribute usage is different. An instance variable can be used directly or indirectly by methods. An instance variable is used directly by a method *M*, if the instance variable appears in the body of the method *M*. The instance variable is indirectly used, if it is directly used by another method *M'* which is invoked directly or indirectly by M. Two methods are directly connected if they use directly or indirectly a common attribute. TCC is defined as the percentage of pairs of methods that are directly connected. LCC counts the pairs of methods that are directly or indirectly connected. Constructors and destructors are not taken into account for computing LCC and TCC. The range of TCC and LCC is always in the [0,1] interval. They propose three ways to calculate TCC and LCC: (1) include inherited methods and inherited instance variables in the analysis, (2) exclude inherited methods and inherited instance variables from the analysis, or (3) exclude inherited methods but include inherited instance variables. With respect to the three ways of calculating their metrics, Bieman and Kang do not express any preference. We opted for evaluating them according to the first way, considering inheritance as an intrinsic facet of OO systems. LCC is an extension of TCC in that additional features are taken into account. LCC being more comprehensive than TCC, we adopted LCC, together with LCOM, as the prime cohesion metrics of our experimentation.

2.2. Impact model

One way of assessing the changeability of an OO system is by performing a change impact analysis. By changeability we mean its capacity to absorb changes. In this study, the changeability of OO software is assessed by an impact model defined in our previous work [3,4]. Below, we detail the changes considered and the links involved, and we introduce the notions of impact and impact expression.

Changes

A change applies to a class, a variable or a method. Examples are deleting a variable, changing the signature of a method, or removing a class from the list of parents of another class. Thirteen changes have been identified:

(i) *Variable*: addition, deletion, type change, and scope of change

(ii) *Method*: addition, deletion, return type change, implementation change, signature change, and scope change

(iii) *Class*: addition, deletion, and structure change.

The changes considered in this paper are atomic changes. More complex changes, for instance refactoring operations such as moving a variable or a method along the class hierarchy, or inserting a new class to factor out some common characteristics of a group of classes, are subject to future work. Among other things, an attempt will be made to define them as a combination of atomic changes. In this way, changes might be dealt with at a higher level of abstraction.

Links

The following links connect classes one to another. They reflect usual connections in OO systems, and are not specific to any particular OO programming language.

S (*association*): one class references variables of another class

G (*aggregation*): the definition of one class involves objects of the other class

H (*inheritance*): one class inherits the features defined in another (parent) class

I (*invocation*): methods in one class invoke methods defined in another class.

L (*local*): pseudo-link meaning that a change in a class may also have an impact in that same class.

The links are independent from each other, and we can expect to find any number and type of links between two classes. Note that instantiation is not a link in its own right, but is taken into account with the invocation link.

Impact

We call *impact of a change* the set of classes that require correction as a result of that change. Our model implies that one single change is considered at a time. The impact depends on two factors. One factor is the type of change. For example, a change to a variable type has an impact on all classes referencing that variable, whereas the addition of a variable has no impact on those classes. Given a type of change, the second factor is the nature of the links involved.[1] If, for instance, the scope of a method is changed from public to protected, the classes that invoke the method will be impacted, with the exception of the derived classes. Note that we limit ourselves to syntactic impact; considering semantic impact, for instance runtime errors, is beyond the scope of this paper. The impact of change ch_j to class cl_i is defined by a set expression **E** in which the variables are the sets defined by the various links:

$$Impact(cl_i, ch_j) = \mathbf{E}(\mathbf{S}, \mathbf{G}, \mathbf{H}, \mathbf{I}, \mathbf{L})$$

For example,

$$Impact(cl_i, ch_j) = \mathbf{SH'+G}$$

means that the impacted classes are those associated (**S**) with, but not inheriting (**H'**) from cl_i or those aggregated (**G**) with cl_i.

2.3. Application to C++

The industrial partner of our project was interested in the evaluation of programs in C++ for which only the code was available. The model was therefore mapped into that language.

In the C++ model, a change is a syntactic change to the code, and impact is considered if, as a result of that change, the code at some other place does not recompile successfully. The links identified in the conceptual model exist at the code level, and an additional one, **F** for *friendship,* is added to reflect the existence of this feature in C++. **F** was not considered as a link at the design level since it is specific to C++. Possible changes were enumerated and for each, the impact set-expression was derived by examining all possible combinations of links between a changed class and another class. As an example, the change in a variable's scope from public to private (code change from `public int v;` to `private int v;`) results in the impact **SF'**, meaning that the impacted classes are those linked to the changed class by association but not by friendship. A total of 66 changes and their impact expressions was compiled, 12 for variables, 35 for methods, and 19 for classes (see [3] for more details about the list of changes and impact calculations).

[1] Please note that the impact of a change does depend only on the two factors described above. There is no relationship between the impacts of two different changes.

3. Empirical validation of cohesion-changeability relationship

3.1. Objectives

One way to assess the changeability of a software system is to find some design properties that can be used as changeability indicators. In the realm of OO systems, experiments have been conducted showing that coupling between classes is an indicator of changeability. Chaumun et al. observed a high correlation between changeability and some coupling metrics, across different industrial systems and across various types of changes [4].

However, measuring coupling is difficult since it is an inter-class property. In fact, to measure it, the knowledge of the whole system and of all links between classes must be mastered.

Cohesion is an intra-class property; to measure it we only need to consider the studied class. Note that a widely held belief in the design community states that high cohesion is related to low coupling. Because of this supposed relationship, we decided to investigate cohesion as a changeability indicator. If this investigation proved successful, the assessment of cohesion as changeability indicator would be less costly that the computation of coupling.

3.2. Experimental procedure

To test the hypothesis that cohesion is correlated to changeability, we adopted the well-known cohesion metrics, LCC and LCOM (see Section 2.1).

Due to lack of resources, we were unable to investigate the whole list of 66 changes of our impact model for C++ (see Section 2.3). Rather, we limited ourselves to six changes, which we selected according to four criteria. First, there should be at least one change for each component (variable, method, and class). Second, a selected change should indeed have an impact in at least one other class (according to our model, there are 29 changes with no such impact). Third, the impact expression should be

different for any pair of changes; since otherwise, we would have obtained duplicate results. And fourth, as an informal criterion, we required the selected changes to be of practical relevance, that is, they should be suitable to be exercised in practice. Table 1 lists the six changes considered and their corresponding impact expression.

Table 1. Investigated changes with impact expressions

	Change	Impact Expression
1.	Variable type change	**S + L**
2.	Variable scope change from public to protected	**SH'F'**
3.	Method signature change	**I + L**
4.	Method scope change from public to protected	**H'IF'**
5.	Class derivation change from public to protected	**H'F' (S + I)**
6.	Addition of abstract class in class inheritance structure	**S + G + H + I + L**

In the experiment, we first extracted the LCC and LCOM metrics from the test systems. Next, for each of the six changes considered, and each of the test systems, we determined its test set, that is, the set of classes for which the change is applicable. For example, when considering the method scope change from public to protected (Change #4), only classes with at least one public method were included in the test set. Then, for each class in each test set, the change impact for the given change was calculated, i.e., the number of classes that would be impacted. If the change was one that affected a variable or method component (Changes #1 through #4), the change impacts for each individual variable or method of the given class were added together, and the total was divided by the number of variables or methods in the class.

Once the metrics and impact data were collected, we investigated the correlation between each change impact and each design metric for all the classes involved in the test sets. Then, in each case the correlation coefficient was calculated.

3.3. Environment

In this section, we first present the three test systems of the experiment. Then, the environment in which the experiment was conducted is described. Finally, we discuss the experimental procedure that was adopted.

Three industrial systems were considered. They vary in class size and application domain. The first test system is *XForms*, which can be freely downloaded from the web [14]. It is a graphical user interface toolkit for X window systems. It is the smallest of the test systems (see Table 1). *ET++*, the second test system, is a well-known application framework [13]. The version used in the experiment is the one included in the SNiFF+ development environment [12]. The third and largest test system was provided by *Bell Canada*, and is called, for confidentiality reasons, *System-B*. It is used for decision making in telecommunications. Table 2 provides some size metrics for these systems. Note that the header files of the programs are included in the numbers shown in the lower part of the table (last six rows), whereas the numbers in the upper part (first four rows) represent the system that was effectively investigated in the study. In *# of classes* many elements such us unions are counted by the compiler as classes.

Table 2. Size metrics of test systems

	XForms	*ET++*	*System-B*
Lines of code	7 117	70 796	291 619
Lines of pure comments	764	3 494	71 209
Blank lines	1 009	12 892	90 426
# of effective classes	83	584	1 226
# of classes	221	722	1 420
# of files (.C/.h)	143	485	1 153
# of generalizations	75	466	941
# of methods	450	6 255	8 594
# of variables	1 928	4 460	13 624
Size in repository	2.9 MB	19.3 MB	41.0 MB

To calculate the metrics involved in the experimentation, we used the SPOOL environment (see Figure 1). This environment has been developed for the entire SPOOL project and comprises various analysis and visualization capabilities to cope with large-scale software systems [10].

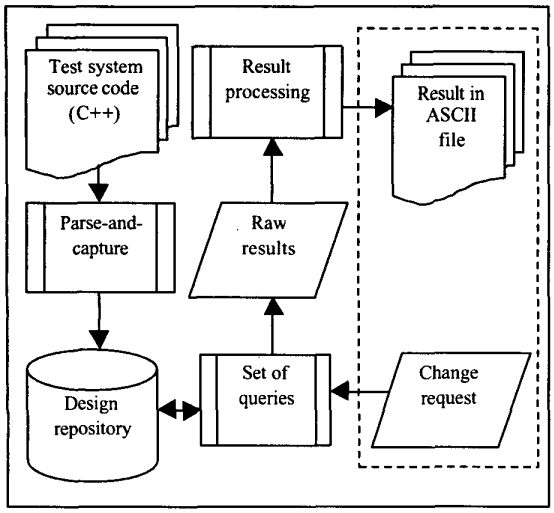

Figure 1. Environment for metrics calculation

The environment provides a repository-based solution. A parsing tool, parses the test system source code. The parsed information contains data about all classes and links in the system. This information is captured and fed into a design repository. Metrics requests are batch-processed using a flexible report generator mechanism. Reports typically contain information on the metrics as well as on the target class, methods, and variables. This triggers a set of queries corresponding to the specified metrics. The code in these queries uses the metrics request information as parameters to interrogate the repository.

We collected cohesion metrics values from the three test systems. Furthermore, we gathered the impact value of the six changes. For each metric and impact involved in the experimentation, we calculated some descriptive statistics (minimum, maximum, mean, median, and standard deviation; see Appendix A and B). To test our hypotheses, we calculated for the two cohesion metrics the Pearson coefficient of correlation in respect to the six impacts of change (see Appendix C).

3.4. Results

Each of the six changes was applied to each test system. The impact values are presented in Appendix A.

The values vary from one system to another, from one change to another, and no general conclusion can be drawn on the impact of a given change. Comparison between changes, however, yields some results. Based on both mean values and median values, a classification of changes by impact comes out. Among the six changes investigated, the most expensive one, across systems, is the addition of an abstract class in the inheritance structure of a class (Change #6). On the other hand, the least expensive one is to change the scope of a method from public to protected (Change #4). This might have been expected, considering their impact expressions (see Appendix A).

According to C&K [6] and Bieman and Kung [2], a class is strongly cohesive when LCC ≈ 1 or LCOM = 0. Appendix B shows the mean value for both LCC and LCOM. Based on the mean value of LCC and LCOM, $\mu(LCC) = 0.62$ and $\mu(LCOM) = 1$, we can conclude that Xforms classes are not so strongly cohesive. For ET++, $\mu(LCC) = 0.42$ and $\mu(LCOM) = 89.70$. Finally for System-B, $\mu(LCC) = 0.56$ and $\mu(LCOM) = 145.73$. Based on these values, and referring to the definition of both LCOM and LCC, we concluded that the three test systems classes are not strongly cohesive.

Note that a similar reasoning can be done based on the median value of both LCC and LCOM.

According to the median of LCOM for the three test systems, half of the classes could be split. On the other hand, the median values of LCC for Xforms (0.69) and System B (0.61) suggest that half of the classes have a LCC value bigger that 0.6. At this stage, we can conclude that there is discrepancy between LCC and LCOM.

The Pearson correlation coefficients are presented in Appendix C. Two exceptions aside, most correlation coefficients for the two cohesion metrics are weak. The two exceptions are, for Xforms, the correlation coefficients between LCC and the change #1 and between LCC and change #5, with values in either case around 0.5. However, they are not significant enough to confirm the correlation hypothesis.

4. Analysis

4.1. Investigation of weakly cohesive classes

The goal of our study was to find a correlation between cohesion and changeability, but the result was negative. Consequently, we set out to investigate this absence of correlation. We came up with the following explanations: (1) the cohesion metrics chosen for the experimentation or the impact of changes are not the right ones, (2) there is no relationship between cohesion and changeability. Explanation (2), being counter to a widely held belief in the design community, was discarded. And, since the changes of the impact model were already validated in [4,5], we focused our investigation on the following sub-hypothesis:

(1A) The LCC and LCOM metrics do not correctly measure cohesion.

Thus, we question the quality of the investigated cohe-

sion metrics (sub-hypothesis (1A)). Intuitively, when they show a high class cohesion (LCC =1 or LCOM = 0), the classes are probably quite cohesive. However, we were doubtful about the expressiveness of LCOM and LCC in the presence of weak class cohesion. Thus, we set out to study manually various weakly cohesive classes occurring in the three test systems.

We chose from each of the three test systems classes that exhibit weak cohesion (LCC < 0.5 and/or/ LCOM > 0), to verify if they were real candidates for splitting. After studying these classes, we found that many of them should not be split. Some classes had no variables or only abstract methods, yielding to low LCC or high LCOM values. We also noticed that for some classes, counting inherited variables or inherited methods reduces the cohesion metric. Some classes have multiple methods that share no variables but perform related functionality's and putting each method in a different class would be against good OO design. Finally, we identified several classes that have numerous attributes for describing internal states, together with an equally large number of methods for manipulating them. These attributes belong together and should not be separated.

Based on this analysis, we notice that low values of LCC and high values of LCOM do not assure a weakly cohesive class. In fact, although many studied classes show a weak cohesiveness based on cohesive metrics, their source code shows an acceptable cohesion. We conclude that as measured, LCC and LCOM do not reflect in general the cohesion property of a class.

4.2. Reasoning about results

The results found in our study call for a revision of the definition of actual cohesion metrics. Chae and Kwon [8] observe that some special methods must be treated in such way as not to compromise the value of the cohesion metrics. They also suggest that cohesion metrics may take into account some additional characteristics of classes, for instance, the patterns of interaction among the members of a class. However, all these additional properties are not trivial to measure, since they are semantic.

We can early conclude that, as long as a new cohesion metric is not defined, taking into account important facets of the cohesion property, actually defined cohesion metrics cannot be trusted as changeability indicators.

5. Conclusion

In this paper, our major goal was to validate cohesion metrics as changeability indicators. To this end, we tried to correlate cohesion metrics with impact of change. First, a model of software changed and change impact was adapted for C++ language. For practical reasons, we only investigated six changes, chosen to be representative of C++ systems changes. Furthermore, we limited our definition of change impact to recompilation errors. As cohesion metrics, we chose LCC and LCOM. Data about these metrics systems were collected on three different industrial. Our experimentation showed a weak correlation between cohesion metrics and changeability.

According to OO design principles, a good design exhibiting high class cohesion goes together with less impact of changes. A relationship should therefore exist between cohesion and changeability. We suspected that the cohesion metrics used in the experimentation do not reflect the real cohesion of a class. We decided to investigate manually classes with low cohesion metric values. We found that although some classes have low LCC and/or high LCOM, these classes are actually cohesive. In fact many facets of the cohesion property are missing in the actual cohesion metrics.

For the same test systems, our previous work [5] showed that coupling is a changeability indicator. However, we could not come up with the same conclusion for cohesion. Thus, we conclude, that actually defined cohesion metrics are not good changeability indicators. As future work, we are trying to feed our database with new test systems. In the same direction, we project to extend the change impact model.

6. References

[1] Lionel C. Briand, John Daly, and Jurgen Wust. A unified framework for cohesion measurement in object-oriented systems. In *Empirical Software Engineering - An International Journal*, 3(1), pages 67-117, 1998.

[2] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the Symposium on Software Reusability (SSR'95)*, pages 259-262, Seattle, WA, April 1995.

[3] M. A. Chaumun. Change impact analysis in object-oriented Systems: Conceptual Model and Application to C++. *Master's thesis*, Université de Montréal, Canada, November 1998.

[4] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, and Francois Lustman. A change impact model for changeability assessment in object-oriented systems. In *Proceedings of the Third Euromicro Working Conference on Software Maintenance and Reengineering*, pages 130-138, Amsterdam, The Netherlands, March 1999.

[5] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, Francois Lustman, and Guy St-Denis. Design properties and object-oriented software changeability. In Proceedings of the Fourth Euromicro Working Conference on Software Maintenance and Reengineering, pages 45-54, Zurich, Switzerland, February 2000. IEEE.

[6] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. In *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pages 476-493, June 1994.

[7] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. In *IEEE Transactions on Software Engineering*, 24(8):629-639, August 1998.

[8] Heung Seok Chae and Yong Rae Kwon. A cohesion measure for classes in object-oriented systems, In *Proceedings of the Fifth international Software Metrics Symposium*, pages 158-166, Bethesda, MD, November 1998.
[9] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. *Proc. Int. Symposium on Applied Corporate Computing*, pages 25-27, October, 1995.
[10] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse engineering of design components. In Proceedings of the Twenty-First International Conference on Software Engineering, pages 226-235, Los Angeles, CA, May 1999. IEEE.
[11] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. In *Journal of Systems and Software*, 23:111-122, February, 1993.
[12]TakeFive GmbH, Salzburg, Austria. SNiFF+ Documentation Set, 1999. Available online at: <http://www.takefive.com>.
[13] Andre Weinand, Erich Gamma, and Rudolf Marty. Design and implementation of ET++, a seamless object-oriented application framework. In *Structured Programming*, 10(2): 63-87, April-June, 1989.
[14] Xforms Library. Graphical user interface for X. Documentation Set, 1997. Available online at <http://bragg.phys.uwm.edu/xforms>.
[15] Edward Yourdon and Larry L. Constantine. Structured Design. Prentice Hall, Englewood Cliffs, N.J., 1979.

Appendix A: Change impact results for the three test systems

	Change	System	# of classes in test set	Min.	Max.	Mean	Median	Std. Dev.
1.	Variable type change	*Xforms*	37	1	20	**1.78**	1	3.17
		ET++	416	1	81	**2.02**	1	5.97
		System-B	707	1	32	**1.46**	1	1.85
2.	Variable scope change from public to protected	*Xforms*	1	-	-	-	-	-
		ET++	65	0	80	**3.78**	0.67 [2]	12
		System-B	72	0	52	**1.84**	1	6.21
3.	Method signature change	*Xforms*	70	1	3.67	**1.19**	1	0.49
		ET++	502	1	17.64	**1.46**	1	1.26
		System-B	1 221	1	38.60	**1.77**	1	2.06
4.	Method scope change from public to protected	*Xforms*	70	0	2.67	**0.18**	0	0.48
		ET++	496	0	16.64	**0.40**	0	1.19
		System-B	1 174	0	37.39	**0.60**	0	1.79
5.	Class derivation change from public to protected	*Xforms*	65	0	4	**0.32**	0	0.89
		ET++	458	0	281	**3.71**	0	16.46
		System-B	1 052	0	291	**4.42**	0	20.03
6.	Addition of abstract class in class inheritance structure	*Xforms*	83	1	64	**4.90**	**1**	11.49
		ET++	584	1	393	**8.84**	**2**	31.16
		System-B	1 226	1	743	**11.34**	**2**	38.20

[2] Note that the impact values are calculated as averages (Section3.2), and hence a median need not to be an integer.

Appendix B: Metrics results for the three test systems

System		LCC	LCOM
Xforms 83 classes	**Minimum**	0	0
	Maximum	1	208
	Mean	0.62	5.81
	Median	0.69	1
	Std. Dev.	0.27	25.40
ET++ 584 classes	**Minimum**	0	0
	Maximum	1	4714
	Mean	0.42	89.07
	Median	0.33	6
	Std. Dev.	0.31	352.81
System-B 1226 classes	**Minimum**	0	0
	Maximum	1	11706
	Mean	0.56	145.73
	Median	0.61	10
	Std. Dev.	0.31	695.72

Appendix C: Correlation coefficients for the three test systems

	Change	System	LCC	LCOM
1.	Variable type change	*Xforms*	-0.52	0.11
		ET++	0.11	0.18
		System-B	-0.06	0.02
2.	Variable scope change from public to protected	*XForms*	[3]	[3]
		ET++	0.31	0.06
		System-B	-0.10	-0.01
3.	Method signature change	*XForms*	-0.44	-0.01
		ET++	-0.02	0.31
		System-B	-0.07	0.21
4.	Method scope change from public to protected	*XForms*	-0.44	0.09
		ET++	0.25	0.12
		System-B	0.01	0.13
5.	Class derivation change from public to protected	*XForms*	-0.52	0.30
		ET++	0.01	0.36
		System-B	-0.01	0.05
6.	Addition of abstract class in class inheri-tance structure	*XForms*	-0.39	-0.01
		ET++	0.03	0.34
		System-B	-0.07	0.35

[3] There is only one class in the test set.