2014 9th International Conference on the Quality of Information and Communications Technology

Estrangement Between Classes: Test Coverage-based Assessment of Coupling Strength Between Pairs of Classes

George Kakarontzas*, Vassilis C. Gerogiannis†, Stamatia Bibi‡, Ioannis Stamelos‡
*Computer Science and Engineering Department
Technological Educational Institute of Thessaly
Larissa, Greece
Email: gkakaron@teilar.gr
†Department of Business Administration
Technological Educational Institute of Thessaly
Larissa, Greece
Email: gerogian@teilar.gr
‡Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
Email: {sbibi,stamelos}@csd.auth.gr

***Abstract*—This work discusses a new metric, *Estrangement Between Classes (EBC)*, that is derived by executing tests. This metric is based on the statement coverage of tests and provides assessment of the strength of associations between classes. We demonstrate with an illustrative example of the popular Apache Email component that this new metric can provide additional information in reverse engineered class diagrams by highlighting missing associations in these diagrams, the strength of existing associations and utility classes. It can also be effective in indicating the important design elements in cases of over-engineered or dead code. The proposed metric can be potentially used in the context of agile methods of software development during refactoring and program maintenance as comprehension aid. Since *EBC* is based on tests, no additional effort is required by developers who follow the Test-Driven approach or generally develop tests.**

I. Introduction

With the extensive use of agile methods [1] and test-driven development [2] it becomes increasingly probable that test suites are not only available but that they also provide adequate coverage of the source code. Extensive test cases with adequate test coverage are important to the functional correctness of the source code. Also approaches, such as the one proposed in [3], use tests for program comprehension activities such as feature location and the results of these approaches are improved when test coverage is improved. In this work we use *test coverage values* to propose a dynamic metric that captures the lack of coupling between two classes and that is calculated based on the statement coverage of the tests. The proposed metric also contains an internal coupling metric component, which can be used independently if a coupling metric is required instead.

To explain the basic idea, assume the situation depicted in Fig. 1. In this example classes A and B are both associated to class C. This is depicted in the UML class diagram at the top of Fig. 1. In addition, instances of classes A and B call the methods m3() and m4() respectively on an instance of class C. This is depicted in the UML sequence diagram at the bottom of Fig. 1. Assuming that this is all the information we have, the coupling between classes (A, C) and between classes (B, C) is exactly the same, since in both cases a static association exists and in both cases a method is called.

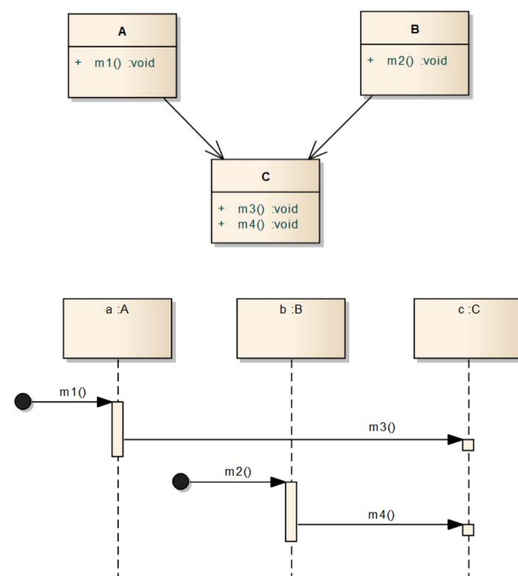

Fig. 1. Example class and sequence diagrams

However, the situation may be different if we examine the number of statements in m3() executed as a result of the first

978-1-4799-6133-7/14 $31.00 © 2014 IEEE

DOI 10.1109/QUATIC.2014.13
IEEE computer society

call and the number of statements in m4() executed as a result of the second call. This information could be very important since m3() may be, for example, a simple getter with one line of code, whereas m4() could be an elaborate service method with tens of lines of code. Assume, for example, that A and B belong to different packages and we consider if we should place C in A's package or in B's package. We would like to take this decision based on the strength of coupling between these two pairs of classes, namely (A, C) and (B, C), and select to place C with the class with which it is coupled more intensely. Static and dynamic coupling measures (e.g., CBO [4] and the suite in [5], respectively), discussed in more detail in the 'Related Work' section, although they highlight coupling they do not provide this information. Intuitively, however, such information is important, since the number of statements executed as a result of messages may be indicative of the significance of the clients' objects dependence on the service or provider objects. For example, in the case of Fig. 1 we could say that class B is more strongly coupled to class C than class A since it uses a larger percentage of class's C statements. So this information could help us decide in which package to place C, if A and B are in different packages and we want to place C together with the most related class to it.

In this paper we explore the idea that statement coverage may provide also an indication of the strength of coupling between classes or the lack of it. Statement coverage is an adequacy criterion in testing [6], [7] in which "the percentage of the statements exercised by testing is a measurement of the adequacy" [6]. Using test coverage of test cases designed for a particular class in other related classes we can have an indication of the coupling strength beyond the simple assertion that two classes are coupled or not. A more accurate estimation of coupling is important during design and comprehension activities.

In Section II we discuss the idea of Estrangement Between Classes (EBC) based on test coverage and introduce the definition of the EBC metric and its internal part, Test Coverage Induced Coupling (TCIC). In Section III we present the results obtained using the EBC metric in the popular Apache Commons Email component. Then, in Section IV we perform an initial comparison of the coupling part of the proposed metric to established coupling metrics. In Section V we discuss related work and in Section VI we conclude the paper and briefly discuss future research directions.

II. Estrangement Between Classes

Assume that we have two classes *A* and *B*. After these classes are integrated in the system, whenever we test class A with the test suite prepared specifically for A, the control may also pass from class B, if A uses B or otherwise depends on it and given that mock objects are not used in the place of all real object collaborators. The extent of this usage may be indicative of the strength of the association that exists between the two classes during runtime.

Figure 2 depicts a few illustrative cases. In all three cases we execute the tests of the class on the left-hand side (A1, A2 and A3) but, after classes' integration, control also passes from classes on the right-hand side (B1, B2 and B3, respectively). In Fig. 2 shaded areas represent the percentage of statements covered by the test suite at each class. Intuitively we can argue that class A2 is more related to class B2 than class A1 is related to class B1. This is because although we achieve the same coverage in both A1 and A2 (the shaded area), the coverage in class B2 is larger than that in class B1. Similarly we can argue that A3 is more related to B3 than A2 is to B2 because the same coverage is achieved in both cases to the right-hand side class (B2 or B3); however, this coverage is achieved in the case of B3 with fewer tests since the coverage in A3 is less extensive compared to the coverage in A2. We assume that if more tests are added to cover more of A3 the coverage to B3 will also grow accordingly.

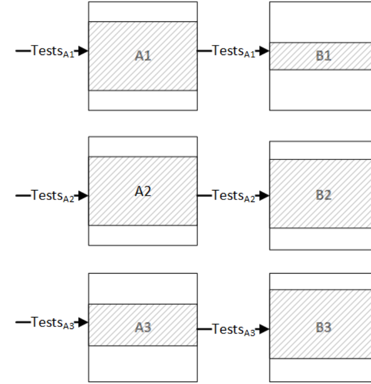

Fig. 2. Coverage graphic examples

To capture this intuition as a metric, we define the strength of an association between two classes A and B as the statement coverage induced in class B by the test suite of class A *divided by* the coverage of A. We call this metric Test Coverage Induced Coupling (TCIC) and is defined as can be seen in Eq. 1. The denominator signifies the extent to which class A is tested by the same test suite (i.e., its own test suite). We use $C_{TS_X}(Y)$ to indicate the statement coverage of test suite of class X in class Y.

$$TCIC(A, B) = \frac{C_{TS_A}(B)}{C_{TS_A}(A)} \quad (1)$$

Then EBC from class A to class B is defined by Eq. 2 :

$$EBC(A, B) = \begin{cases} 1 - TCIC(A, B) & \text{if } TCIC(A, B) \leq 1 \\ 0 & \text{if } TCIC(A, B) > 1. \end{cases} \quad (2)$$

In Eq. 2:

1) We subtract the strength of coupling (as denoted by the TCIC) from 1 to get the estrangement between classes. If two classes are related less, then TCIC will be smaller and EBC will be larger. Thus, EBC takes larger values for less related classes. Furthermore, it takes values

in the range $[0, 1]$. Therefore, it measures the lack of coupling.

2) In Eq. 2 the bottom case is necessary to avoid negative EBC values when the statement coverage in B surpasses that in A. In these cases $C_{TS_A}(B) > C_{TS_A}(A)$. This in turn implies that $TCIC(A, B) > 1.0$ which implies that $1 - TCIC(A, B) < 0$. To restrict the value range for EBC to values in the range $[0, 1]$ and avoid negative values, we assign the value 0 to EBC in such cases. It is worth noticing that these cases can occur in practice if class B contains one or a few methods that are called during the testing of class A. Then, if the test suite of class A does not cover all of A's statements it will be a case in which the test coverage of the test suite of A will be smaller in A than in B. We view EBC as a measure of the lack of the association of a class to another class and we consider the estrangement of a class to itself to be zero and, consequently, we want to restrict its association to other classes to this value range. However, if more precision is required then TCIC values can be used instead.
3) EBC and TCIC are not symmetric since in the general case $EBC(A, B) \neq EBC(B, A)$ and $TCIC(A, B) \neq TCIC(B, A)$.
4) Both EBC and TCIC are relative to the extent that A was tested since B's statement coverage (numerator in TCIC) is divided by A's test coverage (denominator in TCIC). So, for example, if coverage in A is 80% and in B 20% then the ratio is 25% which is a larger number than the coverage in B to compensate that some of the missing relation may be due to not testing A entirely.
5) Most importantly, the proposed metrics can only be effective if tests have a significant test coverage. Otherwise the metrics possibly cannot provide the desired information. For example, if a class is not tested at all, the denominator in TCIC may be zero which makes EBC undefined. This makes these metrics appropriate for agile methods that follow the Test-Driven approach or apply extensively testing to verify the quality of the source code. In general, we can suggest that EBC/TCIC should be used as indicators if $C_{TS_A}(A) \geq 70\%$ although the actual threshold must be determined with empirical studies that we will conduct in our future work. We suggest here using this threshold as a starting point since it is a large value close to what is considered in most cases a sufficient test coverage for test adequacy in practical settings. Notice that the use of statement coverage for our purposes may be less demanding than the usual statement coverage use as indication of testing adequacy for functional correctness. For this latter purpose well-tested code is expected to have a statement coverage over 80%. As Martin Fowler observes "*If you are testing thoughtfully and well, I would expect a coverage percentage in the upper 80s or 90s. I would be suspicious of anything like 100% - it would smell of someone writing tests to make the coverage numbers happy, but not thinking about what they are doing*" [8].

III. EBC Case Study

In this section, we will first examine (in Section III-A) how EBC can be used to highlight the importance of classes' associations. Then we will see (in Section III-B) that EBC can also pinpoint possibly wrong design decisions under certain conditions, by examining the same case study after artificially injecting a 'design bug'.

A. Using EBC to highlight the importance of class associations

In order to illustrate the use of EBC we used existing unit tests of the Apache Commons Email ver. 1.3.2 [9] which is the current release at the time of this writing. Existing tests include tests such as the *HtmlEmailTest* which is designed to test the functionality of the *HtmlEmail* class. This class is used for sending HTML formatted email. Along with this class a number of other classes are also contained in the same component package, namely *org.apache.commons.mail*. Initially, we will only consider classes from this package. Later, in Section IV, we will also consider classes from other packages. The classes of this package with extracts from the comments of their developers describing their functionality are the following:

1) *ByteArrayDataSource*: "This class implements a typed DataSource from an InputStream, a byte array or a String (Deprecated)".
2) *DataSourceResolver*: "Creates a *DataSource* based on an URL".
3) *DefaultAuthenticator*: "This is a very simple authentication object that can be used for any transport needing basic userName and password type authentication".
4) *Email*: "The (abstract) base class for all email messages. This class sets the sender's email & name, receiver's email & name, subject, and the sent date. Subclasses are responsible for setting the message body".
5) *EmailAttachment*: "This class models an email attachment. Used by *MultiPartEmail*".
6) *EmailConstants*: "Constants used by Email classes".
7) *EmailException*: "Exception thrown when a checked error occurs in commons-email".
8) *EmailUtils*: "Utility methods used by commons-email".
9) *HtmlEmail*: "This class is used to send HTML formatted email...This class also inherits from *MultiPartEmail*, so it is easy to add attachments to the email".
10) *ImageHtmlEmail*: "Small wrapper class on top of *HtmlEmail* which encapsulates the required logic to retrieve images".
11) *MultiPartEmail*: "This class is used to send multi-part Internet email like messages with attachments".
12) *SimpleEmail*: "This class is used to send simple Internet email messages without attachments".

First we executed (*HtmlEmaiTest*) which is a JUnit test case for *HtmlEmail* class constructed by the component original developers. The test statement coverage of this test case for

the *HtmlEmail* class was not 100% but 74.2%, whereas for another class, namely the *DefaultAuthenticator*, the coverage was 100%. This is because the *DefaultAuthenticator* has only one method that is called during the test. As we mentioned already, cases like this necessitate the bottom branch of Eq. 2 for the avoidance of negative EBC values since $TCIC > 1.0$. In addition, classes *EmailAttachment* and *EmailUtils* also present higher test coverage values than the *HtmlEmail* class with 84% and 82.9% statement coverage, respectively.

We have used in this work EclEmma [10] as the test coverage tool in the Eclipse IDE. EclEmma provides test coverage for JUnit [11] tests and reports this coverage as well as provides coloring for the statements that were covered during the execution of a test suite.

All the results of running the *HtmlEmailTest* and the EBC of each class to the *HtmlEmail* class are shown in Table I. In this table (as well as in Tables II, III, IV and V) we use a gray row background to indicate which is the source class of EBC (i.e., the class under test). EBC values reported at each table are the EBC values for this source class with the other classes used as targets. Notice that the results are sorted in increasing EBC order. There are a number of classes that present zero estrangement to the *HtmlEmail* class. These classes have in fact different TCIC values (see Tables VII and VIII for the details), however their coverage is more than the class under test and, therefore, according to the bottom case of Eq. 2 their EBC value is zero. If more discrimination power between the group of classes that have EBC value of zero is needed, then the TCIC values can be used instead.

TABLE I
EBC BETWEEN HTMLEMAIL AND OTHER CLASSES BASED ON COVERAGE OF HTMLEMAILTEST

Class	Missed	Covered	Coverage	EBC
DefaultAuthenticator	0	13	100.00%	0.00%
EmailAttachment	8	42	84.00%	0.00%
EmailUtils	49	237	82.90%	0.00%
HtmlEmail	165	475	74.20%	0.00%
Email	916	462	33.50%	54.85%
MultiPartEmail	305	131	30.00%	59.57%
EmailException	47	9	16.10%	78.30%
ByteArrayDataSource	204	0	0.00%	100.00%
EmailConstants	3	0	0.00%	100.00%
ImageHtmlEmail	149	0	0.00%	100.00%
SimpleEmail	17	0	0.00%	100.00%

According to the results in Table I, the most estranged classes to the *HTMLEmail* class are *SimpleEmail*, *ImageHtmlEmail*, *EmailConstants* and *ByteArrayDataSource* classes. Indeed this make sense, since the *SimpleEmail* class is an alternative class to send emails (simple emails and not multipart HTML emails) where the *ByteArrayDataSource* class, according to its description, is a typed data source and not specific to the HTML Email core functionality and, furthermore, it is also deprecated in the tested release. Class *EmailConstants* contains only static fields and no methods and *ImageHtmlEmail* is a subclass of *HTMLEmail* (see Fig. 3) and, therefore, is not referenced in the tests of its parent class.

Using the *SimpleEmailTest* which tests the *SimpleEmail* class, we get a completely different picture, since some classes that were covered completely (e.g., *EmailAttachment, HtmlEmail*) or to a large extent (e.g., *MultiPartEmail*) by the *HtmlEmailTest* are now completely missed. More specifically, the *SimpleEmail* class is covered 100% along with the *DefaultAuthenticator*. The *Email*, *EmailUtils* and *EmailException* classes are also covered to a lesser extent. On the other hand, the HTML-Multipart email group of classes are estranged to the Simple email class which is a different type of email. The results of running the *SimpleEmailTest* and the EBC of each class to the target *SimpleEmail* class are shown in Table II.

TABLE II
EBC BETWEEN SIMPLEEMAIL AND OTHER CLASSES BASED ON COVERAGE OF SIMPLEEMAILTEST

Class	Missed	Covered	Coverage	EBC
DefaultAuthenticator	0	13	100.00%	0.00%
SimpleEmail	0	17	100.00%	0.00%
Email	850	528	38.30%	61.70%
EmailUtils	185	101	35.30%	64.70%
EmailException	52	4	7.10%	92.90%
ByteArrayDataSource	204	0	0.00%	100.00%
EmailAttachment	50	0	0.00%	100.00%
EmailConstants	3	0	0.00%	100.00%
HtmlEmail	640	0	0.00%	100.00%
ImageHtmlEmail	149	0	0.00%	100.00%
MultiPartEmail	436	0	0.00%	100.00%

Interestingly, the above mentioned partition in two separate groups of classes (i.e., the MutliPart-HtmlEmail group and the SimpleEmail group) is also evident by analyzing the source code statically and generating the UML class diagram with a static analysis tool, as depicted in Fig. 3. In this diagram we can see clearly that there is an inheritance hierarchy with the *Email* base class as a root and two alternative classes for sending email: the *SimpleEmail* class for email messages without attachments and the *MultiPartEmail* for email messages with attachments. Furthermore, the *MultiPartEmail* class is extended by *HtmlEmail* which handles HTML formatted emails. *HtmlEmail*, in turn, is extended by *ImageHtmlEmail* which handles the retrieval of images in HTML formatted emails.

To investigate the relationship between two subclasses of the *MultiPartEmail* hierarchy of Fig. 3, we also executed the *MultiPartEmailTest* in addition to the *HtmlEmailTest*, shown in Table I. *MultiPartEmailTest* is the JUnit test suite for the *MultiPartEmail* class and executing it yields the EBC results shown in Table III.

From the above-mentioned tests we observe the following:

- Class *EmailAttachment* is only a dependency of class *MultiPartEmail*. However EBC reveals that this class is very important both for *MultiPartEmail* (see Table III) as well as for *HtmlEmail* (see Table I), since it is estranged with both of them by 0.00%. On the other hand, the parent class of both these classes, namely *Email* is estranged to class *HtmlEmail* by 54.85% (Table I) and to *MultiPartEmail* by 52.57% (Table III). Therefore,

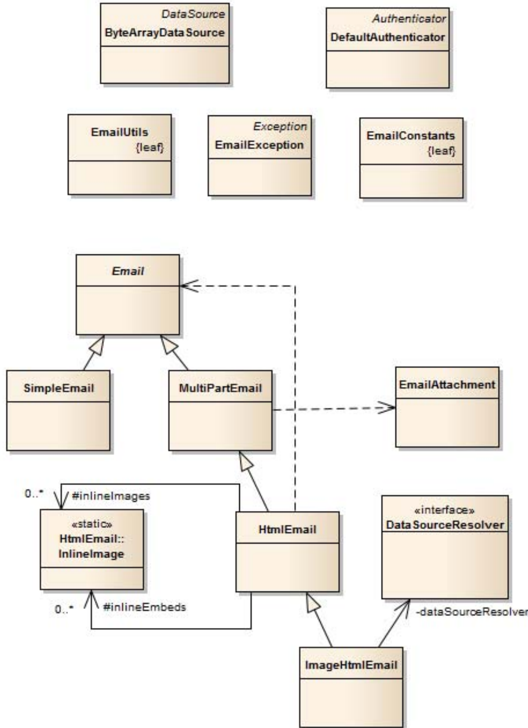

Fig. 3. UML Class Diagram of the email component

TABLE III
EBC BETWEEN MUTLIPARTEMAIL AND OTHER CLASSES BASED ON COVERAGE OF MULTIPARTEMAILTEST

Class	Missed	Covered	Coverage	EBC
DefaultAuthenticator	0	13	100.00%	0.00%
EmailAttachment	0	50	100.00%	0.00%
MultiPartEmail	72	364	83.50%	0.00%
Email	833	545	39.60%	52.57%
EmailUtils	185	101	35.30%	57.72%
EmailException	47	9	16.10%	80.72%
ByteArrayDataSource	204	0	0.00%	100.00%
EmailConstants	3	0	0.00%	100.00%
HtmlEmail	640	0	0.00%	100.00%
ImageHtmlEmail	149	0	0.00%	100.00%
SimpleEmail	17	0	0.00%	100.00%

although inheritance is a stronger association than (UML) dependency, EBC highlights that *this specific dependency* with *EmailAttachment* is important. Indeed the distinguishing characteristic of the Multipart hierarchy of classes in Fig. 3 is that they have attachments, as opposed to Simple emails that do not. *EBC provides, therefore, evidence for the strength of the associations between classes signifying which are more specific to certain groups of classes.*

- Some classes are used mildly by many other classes. For example, *HtmlEmail*, *SimpleEmail* and *MultiPartEmail* all are related to some extent to the classes *EmailException* and *EmailUtils*. This suggests that these classes could be utilities or serving some other general purpose such as exception handling. Of course this is evident from the naming of these two classes as well. However, EBC would have revealed this even if the naming decisions were different. Notice that these classes, namely *EmailException* and *EmailUtils* and others, seem unassociated to other classes in the generated UML diagram in Fig. 3. This is because these classes are not used as class variables, method parameters or method return values. For example, they may be used only as local variables in methods' bodies. Then, some UML tools may not report the dependencies. EBC does not only provide the dependency but also reports its relative strength to the tested class.
- Some classes are not used at all. For example, class *ByteArrayDataSource* is completely estranged to all tested classes in our case study (see Tables I, II and III). This is of course a strong indication of dead code. Indeed this class has been deprecated in the tested version and another class is used instead.
- One class in our case study, namely *DefaultAuthenticator*, has $EBC = 0.0\%$ to all three tested classes (namely *HtmlEmail*, *SimpleEmail* and *MultiPartEmail* in tables I, II and III). This signifies a utility class that is used from most other classes in the system and, at the same time, it is easy to cover entirely. Indeed this class contains only a public constructor that is used by the *Email* base class and, thus, affects all the subclasses.

Examining the coverage in relation to the EBC value in the various tables discussed in this Section, one could argue that EBC does not provide much more information than test coverage. However, the scale imposed by EBC in our opinion is useful as a possible countermeasure to the fact that test coverage varies and is not always 100%. If we consider only one source class and its relation to other classes this is not evident, since the coverage of the source class is always the same. However, if we consider two different source classes and the same target class (e.g., the example in Fig. 1) then we need to incorporate the extent of the test coverage in the source class and EBC attempts to do just that.

B. Using EBC to highlight wrong design decisions

Considering the example, EBC is effective in providing additional information in the static analysis diagrams. However, large and complex systems present additional challenges such as dead or over-engineered code. The proposed measure of estrangement can help in such cases as well.

To demonstrate the effectiveness of estrangement in the face of wrong design decisions we refactored the original source code of the email component by placing an abstract method in the *Email* base class. This method is already implemented in the *MutliPartEmail* class, it concerns email attachments and is the *attach* method. The abstract method placed in the *Email* has an identical signature with the *MutliPartEmail* method:

public abstract MultiPartEmail attach(EmailAttachment attachment) throws EmailException;

This so-called “Pull Up” refactoring is depicted in Fig. 4. This refactoring does not affect the *HtmlEmail* class which already inherits this method by *MultiPartEmail* class, but it affects the *SimpleEmail* class which must now implement this method (although the implementation is empty) in order to be syntactically valid.

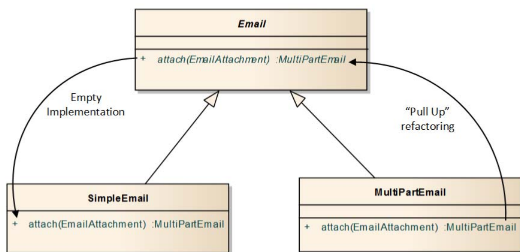

Fig. 4. Pull up refactoring

By pushing its signature to the base *Email* class and by introducing an empty implementation of the method in the *SimpleEmail* class, we now have the static analysis generated UML class diagram depicted in Fig. 5.

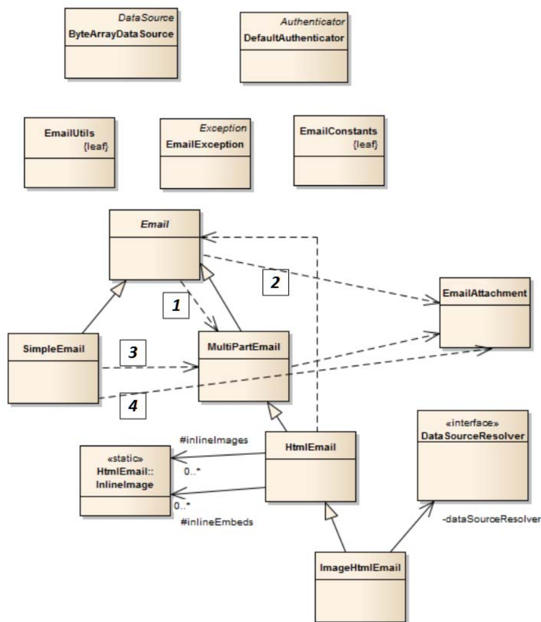

Fig. 5. UML Class Diagram of the email component after introducing a wrong design decision

Notice that four additional dependencies were introduced in the diagram of Fig. 5 compared to the diagram in Fig. 3, since both *Email* and *SimpleEmail* now contain an abstract declaration and an empty implementation, respectively, of the method *attach*. Due to this method’s signature, they both depend now to *EmailAttachment* which is a parameter of *attach* as well as to *MultiPartEmail* which is the return type of *attach*. In Fig. 5 we have numbered these additional dependencies. As a result of these newly introduced dependencies, examining the diagram now does not reveal the cluster of classes where *EmailAttachement* belongs. In fact it is not even evident that we have two separate groups of classes anymore, since the *SimpleEmail* class now depends on the *EmailAttachment* and *MultiPartEmail* classes as well. However, we only refactored the code; Hence we can execute the same tests as before and measure EBC again, since by definition refactoring does not affect functionality (i.e., the original tests should still run successfully). Notice however, that we also had to modify slightly the *MockEmailConcrete* class which extends *Email* and is used for the tests.

The *HtmlEmailTest* and *SimpleEmailTest* were executed successfully and Tables IV and V contrast the original results with the results obtained from the refactored component.

Regarding the *HtmlEmail* test in Table IV, the results are exactly the same as the original results and regarding the *SimpleEmail* test in Table V, the results are (almost) the same as the original results. The small difference in *SimpleEmail’s* EBC values are due to the fact that since the new (null) method does not get executed, test coverage of this class drops slightly and, therefore, EBC in three classes, namely *Email*, *EmailUtils* and *EmailException*, drops slightly as well, although their test coverage is the same.

Consequently, the tests still reveal the dichotomy between the *SimpleEmail* and the *HtmlEmail* classes although this dichotomy is not evident anymore by examining the static structure of the reverse engineering component in Fig. 5. Since the newly introduced implementation of the abstract method in the *SimpleEmail* class, is not executed by the pre-existing *SimpleEmailTest* class the coverage is slightly less than 100% and, therefore, the estrangement is slightly reduced for the classes that are not estranged to the *SimpleEmail* class (since this class coverage is in the denominator of their EBC values).

Of course EBC will be effective if the over-engineered code is not tested. If the over-engineered code is tested then it will be indistinguishable from the rest of the program and both EBC and UML static diagrams will not differentiate it. But in many cases code like this is inserted in the program as a placeholder for future extensions and it is not entirely implemented or tested. In such cases, EBC will be effective in highlighting the unused code. Thus, EBC can highlight possible violations of the so-called YAGNI (You’re NOT gonna need it) principle of Extreme Programming which advises to “Always implement things when you actually need them, never when you just foresee that you need them” [12].

IV. Comparison to Established Coupling Metrics

In this Section, we first define TCIC for a class irrespective to the classes that it uses and then describe how to compute coupling at the level of the system using TCIC. Then, we compare TCIC with established static coupling metrics by performing a correlation analysis. Notice that since both static metrics that we use for comparison are coupling metrics, we use the coupling component of EBC, namely TCIC, which is defined in Eq. 1. Since EBC in Eq. 2, is merely the

TABLE IV
HTMLEMAIL TEST ON REFACTORED EMAIL COMPONENT

	Original		Refactored	
CLASS	Coverage	EBC	Coverage	EBC
DefaultAuthenticator	100.00%	0.00%	100.00%	0.00%
EmailAttachment	84.00%	0.00%	84.00%	0.00%
EmailUtils	82.90%	0.00%	82.90%	0.00%
HtmlEmail	74.20%	0.00%	74.20%	0.00%
Email	33.50%	54.85%	33.50%	54.85%
MultiPartEmail	30.00%	59.57%	30.00%	59.57%
EmailException	16.10%	78.30%	16.10%	78.30%
ByteArrayDataSource	0.00%	100.00%	0.00%	100.00%
EmailConstants	0.00%	100.00%	0.00%	100.00%
ImageHtmlEmail	0.00%	100.00%	0.00%	100.00%
SimpleEmail	0.00%	100.00%	0.00%	100.00%

TABLE V
SIMPLEEMAIL TEST ON REFACTORED EMAIL COMPONENT

	Original		Refactored	
CLASS	Coverage	EBC	Coverage	EBC
DefaultAuthenticator	100.00%	0.00%	100.00%	0.00%
SimpleEmail	100.00%	0.00%	89.50%	0.00%
Email	38.30%	61.70%	38.30%	57.21%
EmailUtils	35.30%	64.70%	35.30%	60.56%
EmailException	7.10%	92.90%	7.10%	92.07%
ByteArrayDataSource	0.00%	100.00%	0.00%	100.00%
EmailAttachment	0.00%	100.00%	0.00%	100.00%
EmailConstants	0.00%	100.00%	0.00%	100.00%
HtmlEmail	0.00%	100.00%	0.00%	100.00%
ImageHtmlEmail	0.00%	100.00%	0.00%	100.00%
MultiPartEmail	0.00%	100.00%	0.00%	100.00%

complement of 1 of TCIC, in most cases the correlations discussed in this section are also valid for EBC. However, in the correlation analysis reported the result would be equally strong but negative rather than positive, since estrangement points to the opposite direction in relation to a coupling metric.

EBC and TCIC are coupling metrics related to pairs of classes. However, it is usual that sometimes we will want to assess coupling or the lack of coupling for a class irrespective of the classes that it uses and at the system level as a whole. In such cases, it would be useful to use the sum of TCICs between pairs of a class with other classes for the total coupling of a class and then use the average of all system classes' individual TCIC values as a metric for the coupling in the system. Equations 3 and 4 define TCIC for a class *A* and for a system *S*, respectively. In these equations, $Classes(S)$ represents the set of all classes in the system.

$$TCIC(A) = \sum\nolimits_{X_i \in Classes(S)} TCIC(A, X_i) \qquad (3)$$

$$TCIC(S) = \frac{\sum_{X_i \in Classes(S)} TCIC(X_i)}{|Classes(S)|} \qquad (4)$$

An interesting question to answer is how TCIC for a class is related to other more established coupling metrics in relation to coupling. In this work we used two such metrics to perform our comparison:

- Message Passing Coupling (MPC): MPC is a metric that captures the count of method calls from methods of a class to methods of other classes [13]. Since TCIC is also based on the coverage of method calls from tests of a class to methods of other classes we assume that a positive correlation may exist between MPC and TCIC.
- Coupling Between Objects (CBO): CBO [4] is a metric that captures the count of other classes a class is coupled to. Again a positive correlation is assumed to exist between CBO and TCIC.

Table VI contains the MPC and CBO values of various classes of the Apache Commons Email component [9]. Together with these two metrics for each of the classes we have also calculated the sum of TCIC values for each class with the other classes using Eq. 3. In Tables VII and VIII, the classes at the first column are used as the source class of which the tests we execute to measure TCIC and EBC with the other classes mentioned in the columns. The TCIC value mentioned in Table VI is the sum of the values of TCIC for each respective source class from Eq. 3. Notice that here we are using classes from two other packages of the commons email component: the class *MimeMessageParser* from the package *org.apache.commons.mail.util* and also the various resolver classes from the package *org.apache.commons.mail.resolver*. Also notice that in Table VI TCIC's higher value is for the *ImageHtmlEmail* class, whereas MPC's and CBO's higher values are for the *HtmlEmail* class. We will provide an explanation for this later on.

TABLE VI
TCIC COMPARED TO ESTABLISHED COUPLING METRICS

Class	MPC	CBO	TCIC(Class)
DefaultAuthenticator	0	0	1.00
Email	29	4	2.98
EmailAttachment	0	0	1.00
EmailUtils	1	1	1.00
HtmlEmail	35	7	6.37
ImageHtmlEmail	6	4	8.48
MultiPartEmail	23	4	4.49
SimpleEmail	3	4	2.81
MimeMessageParser	0	0	1.00
DataSourceClassPathResolver	4	1	1.00
DataSourceCompositeResolver	3	2	2.70
DataSourceFileResolver	3	1	1.00
DataSourceUrlResolver	6	1	1.00

We calculated the non-parametric Spearman correlation ρ for $(MPC(X), TCIC(X)), \forall X \in Classes(S)$ and for $(CBO(X), TCIC(X)), \forall X \in Classes(S)$ to highlight the relationship between these established static metrics of coupling and the proposed coupling metric in the current work.

TABLE IX
SPEARMAN'S ρ WITH STATIC COUPLING METRICS

	Spearman's ρ	P-Value
CBO	0.905426	0.00002061
MPC	0.722784	0.005251

Table IX reports the ρ value and the p-value for these two tests. As can be seen in table IX, there is a statistically significant positive correlation between $(MPC, TCIC)$ and

TABLE VII
TCIC AND EBC FOR APACHE COMMONS EMAIL COMPONENT (PART A)

TARGET	*ByteArrayDataSource*		*DefaultAuthenticator*		*Email*		*EmailAttachment*		*EmailConstants*		*EmailException*		*EmailUtils*		*HtmlEmail*	
SOURCE	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC
DefaultAuthenticator	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
Email	0.00	1.00	1.32	0.00	1.00	0.00	0.00	1.00	0.00	1.00	0.21	0.79	0.45	0.55	0.00	1.00
EmailAttachment	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
EmailUtils	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00
HtmlEmail	0.00	1.00	1.35	0.00	0.45	0.55	1.13	0.00	0.00	1.00	0.22	0.78	1.12	0.00	1.00	0.00
ImageHtmlEmail	0.00	1.00	1.10	0.00	0.37	0.63	0.93	0.07	0.00	1.00	0.18	0.82	0.92	0.08	0.82	0.18
MultiPartEmail	0.00	1.00	1.20	0.00	0.47	0.53	1.20	0.00	0.00	1.00	0.19	0.81	0.42	0.58	0.00	1.00
SimpleEmail	0.00	1.00	1.00	0.00	0.38	0.62	0.00	1.00	0.00	1.00	0.07	0.93	0.35	0.65	0.00	1.00
MimeMessageParser	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
DataSourceClassPathResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
DataSourceCompositeResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
DataSourceFileResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
DataSourceUrlResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00

TABLE VIII
TCIC AND EBC FOR APACHE COMMONS EMAIL COMPONENT (PART B)

TARGET	*ImageHtmlEmail*		*MultiPartEmail*		*SimpleEmail*		*MimeMessageParser*		*DataSourceClass PathResolver*		*DataSource CompositeResolver*		*DataSource FileResolver*		*DataSource UrlResolver*	
SOURCE	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC	TCIC	EBC
DefaultAuthenticator	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
Email	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
EmailAttachment	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
EmailUtils	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
HtmlEmail	0.00	1.00	0.40	0.60	0.00	1.00	0.70	0.30	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
ImageHtmlEmail	1.00	0.00	0.33	0.67	0.00	1.00	0.65	0.35	0.64	0.36	0.67	0.33	0.00	1.00	0.88	0.12
MultiPartEmail	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
SimpleEmail	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
MimeMessageParser	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
DataSourceClassPathResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00
DataSourceCompositeResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.81	0.19	1.00	0.00	0.00	1.00	0.90	0.10
DataSourceFileResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00
DataSourceUrlResolver	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00

$(CBO, TCIC)$, indicating that both MPC and CBO tend to increase simultaneously with TCIC, although due to the small size we believe that this issue deserves further investigation. In the future we plan to perform an experiment with more cases to establish this preliminary result.

Another important observation in relation to the values reported in Tables VII and VIII, has to do with the packages and package dependencies. In relation to this, notice that the classes from the *org.apache.commons.mail.resolver* package are totally estranged to all other classes outside their package. This can be seen by examining the four last rows in Tables VII and VIII which are based on tests of the various resolver classes in the *org.apache.commons.mail.resolver* package. In these rows we can see that they contain $EBC = 1.0$ and $TCIC = 0.0$ for all the classes, except for the last four columns in Table VIII which contain the values with target classes in the same package. Therefore, this shows that this package is independent to other packages and it does not use their services.

On the contrary, examining the first nine rows with classes outside the resolver package in relation to the last four columns in Table VIII we can see that the only class that uses the services of the various resolvers is the *ImageHtmlEmail* class (the sixth row in Table VIII). This class is related according to its test suite to some extent with three out of four resolvers. More specifically: EBC(ImageHtmlEmail, DataSourceClassPathResolver) = 0.36, EBC(ImageHtmlEmail, DataSourceCompositeResolver) = 0.33 and EBC(ImageHtmlEmail, DataSourceUrlResolver) = 0.12. So, there is a dependency from the package *org.apache.commons.mail* to the *org.apache.commons.mail.resolver* package due to the usage of the resolver classes from the *ImageHtmlEmail* class. Notice that this dependency is difficult to establish using static analysis alone. Indeed we can see that the class diagram in Fig. 3 depicts the dependency of the *ImageHtmlEmail* class to the interface *DataSourceResolver*. This interface is implemented by the classes in the resolver package, but the actual dependency with specific resolvers is not captured by the diagram in Fig. 3. Examining the code of the *ImageHtmlEmail* class however clarifies the relationship. As the comments in the beginning of source code file mention: *"The image loading is done by an instance of DataSourceResolver which has to be provided by the caller"*. The actual setting of the resolver by the users of the *ImageHtmlEmail* class is carried out using the method *ImageHtmlEmail::setDataSourceResolver* which accepts as a parameter a resolver instance which is the instance of a class implementing the *DataSourceResolver* interface, as shown in the following code segment.

```
private DataSourceResolver
            dataSourceResolver;
            ...
 public void setDataSourceResolver
   (DataSourceResolver dataSourceResolver)
 {
  this.dataSourceResolver =
                  dataSourceResolver;
 }
```

Therefore, static analysis only reveals the dependency to the *apparent type* which is the *DataSourceResolver* interface. Dynamic analysis however, establishes the *actual type* of the dependency to the specific resolvers used in testing the *ImageHtmlEmail* class. During maintenance it is rather useful to find the *actual types* of the dependencies vs. the *apparent*

types, since it helps highlighting the actual objects involved in the various scenarios during program execution. Here we use the terms actual and apparent type as defined by Liskov in [14], apparent type being the type of the object that the compiler can deduce from declarations and actual type being the subtype of the apparent type that the object actually has during runtime. In fact, by examining Table VI, we can see that both MPC and CBO obtain their highest value for the *HtmlEmail* class. Although the TCIC value for this class is the second highest, the highest TCIC value is obtained for the *ImageHtmlEmail* class which is a subclass of *HtmlEmail* and carries all of the coupling of its parent class but introduces additional coupling. This difference between static metrics and TCIC is due to the incorporation of coupling with the actual types during runtime in TCIC. Indeed by summing up TCIC values for the last four columns of the *ImageHtmlEmail* class in Table VIII, which concern the actual *DataSourceResolvers* used, we get an increase of TCIC by 2.19, which makes $TCIC(ImageHtmlEmail) > TCIC(HtmlEmail)$.

V. Related Work

In the context of program comprehension, feature location approaches based on dynamic execution of tests are an important class. A characteristic example is the work in [3]. In this work the authors propose Spectrum-based Feature Detection (SFC) which uses fault localization techniques for feature location. Their approach is based on the similarity of component vectors touched by runs and evolution vectors with values indicating which runs participate in a feature. The similarity with the component vectors signifies the participation of a component in the feature under question. The similarity to our approach is that it also requires extensive coverage. But our approach is concerned with the association between pairs of classes and not with feature location and also uses directly test coverage values as a way to measure (the lack of) coupling between pairs of classes and not merely improves with better test coverage.

The work described here can therefore be best described as a dynamic metric for assessing the lack of coupling that exists between classes of an Object-Oriented system. Larger EBC values denote classes that are related less. If what is needed instead is a coupling metric then the TCIC component of EBC can be used.

The traditional way to measure coupling was for decades, and still is, the use of static coupling and the most popular metric in this category is the Coupling Between Objects (CBO) metric proposed by Chidamber & Kemerer in their seminal work [4]. However, static coupling is not accurate in the presence of inheritance and polymorphism since it cannot capture the actual classes of which instances are involved. Therefore, the use of dynamic coupling may be more accurate in modern Object-Oriented systems and this is why the use of dynamic metrics to measure coupling is more recent.

Tahir et al. [15] provide a recent systematic mapping study on dynamic metrics and software quality. They found a total of 60 papers from January 1992 until June 2011. Specifically for dynamic coupling metrics they report 25 papers in the same period. They observe that most works in dynamic coupling are motivated by the popular Coupling Between Objects (CBO) static analysis metrics of Chidamber & Kemerer [4]. According to [15], most dynamic metrics are concerned with maintainability and complexity, they measure coupling, cohesion and polymorphism and they focus on Object-Oriented systems. Our proposed metric also measures (the lack of) coupling and also focuses on Object-Oriented systems and can be used during maintainability but also during development as a design aid.

Arisholm et al. [5] present a suite of dynamic metrics. These metrics measure the coupling between objects and classes by measuring the number of distinct messages, the number of distinct method calls and the number of distinct classes involved in the examined scope. Also they include both import and export coupling. Import coupling measures the coupling from the user side, whereas export coupling measures coupling from the provider side. The suite of the proposed metrics in [5] measures set cardinalities of messages, methods and classes but does not assign weights at the members of these sets. For example, all messages are considered equal although some may result in executing hundreds of lines of code and other in executing a few lines. Thus, our proposed metric can be considered complementary to those proposed by [5], since it can provide an assessment of the strength of the association between classes.

Mitchell and Power [16] define the Run-time Coupling Between Objects (RCBO) metric which measures how many classes are accessed by a class during program run. This is therefore a metric between classes and, as the authors in [16] observe, is related to the static CBO metric [4] which measures the number of classes that *could be accessed* by a class during runtime. They also studied the variability in different classes accessed from objects of the same class during runtime. They have concluded that, in some cases, objects of the same class may access different clusters of classes at runtime. Our approach in using test suites arguably aggregates the behavior of objects that belong to the class under test since it contains many expected execution scenarios.

Mitchell and Power again in [17] investigated the influence of instruction coverage on the relationship between static and dynamic coupling metrics. They report that "coverage results have a significant influence on the relationship and thus should always be a measured, recorded factor in any such comparison". The dynamic metrics investigated were the six class metrics proposed by Arisholm et al. in [5]. Initially the authors show, using Principal Component Analysis (PCA) that the dynamic metrics proposed in [5] provide additional information than the CBO metric [4] and they are not surrogate metrics compared to CBO. Then they used statistical regression in which each one of the six dynamic metrics were used in turn as dependent variables and CBO alone and also along with Instruction Coverage were used as the independent variables. In 4 of the six dynamic metrics proposed by [5] there was a significant improvement in R^2 (more than 20%) in

most programs with instruction coverage and CBO together as opposed to CBO alone. This shows that instruction coverage is a decisive explaining factor for dynamic metrics. In the current work we formulate a metric based on test coverage and use this directly as an indicator of the coupling between classes or the lack of it.

Another recent survey on dynamic coupling metrics is provided in [18]. This paper concludes that “the dynamic coupling metrics domain is still quite young in the field of software engineering and faces a number of research challenges in terms of empirical validation and relationship with external software quality attributes”. They have identified 34 metrics from 8 research groups. Some of these metrics originate from the same suite. For example, the 12 metrics of Arisholm et al. [5]. None of the metrics described in [18] use test coverage as a criterion to decide coupling of objects.

VI. Conclusions and Future Research Directions

We proposed Estrangement Between Classes (EBC) and Test Coverage Induced Coupling (TCIC), two measures based on test coverage, as tools for understanding the strength of associations in existing systems and for assisting in designing new systems in the context of an agile development process based on tests. The proposed metrics can compare the strength of associations in statically generated class diagrams or highlight missing associations in such diagrams. They can also be used to detect deprecated or unused code and to highlight utilities and general purpose classes. The proposed method is shown to be effective, under some conditions, in the face of wrong design decisions with an example of our case study. The proposed method and metric can be used without any additional cost in the context of an agile development process in which tests are already constructed, given that mock objects are not used in place of all collaborators, at least not after integrating the various classes.

Future steps include the investigation of additional coverage measures such as branch and path coverage and the empirical validation of the proposed metric in a number of projects, since the use of a single case study can be considered a threat to validity for the generalization of results. Specifically, we are interested in studying the relationship of EBC/TCIC with change proneness for large projects. Also a very important issue is the determination of the threshold of test coverage required to make the proposed metric useful. We suggested a threshold of 70% here but it is important to determine this value empirically. Another interesting idea is to investigate if the frequency of method calls, an aspect not captured directly by EBC and TCIC, can improve the proposed metrics. Since EBC and TCIC are based on test coverage, the number of times a method is called in a test suite is related more to the complexity of the method itself (e.g., the number and types of method’s parameters) rather than the relationship strength between the caller and the method being called. However, EBC and TCIC could be enhanced by statistics of real usage after the program has been put to production in which call frequencies may be indicative of the association strength between classes. Such frequency statistics may prove to be useful in the context of a wider comprehension framework in tandem with EBC and TCIC.

Finally, an important step is to validate the proposed TCIC metric using the five coupling properties proposed in [19].

Acknowledgment

This research has been co-financed by the European Union (European Social Fund) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: ARCHIMEDES III. Investing in knowledge society through the European Social Fund, under the “SPRINT SMEs” project (Research in software process improvement methodologies for Greek small medium sized software development enterprises).

We would also like to thank the anonymous reviewers for providing comments that helped us improve the paper.

References

[1] D. Cohen,M. Lindvall and P. Costa, *An Introduction to Agile Methods*, Advances in Computers, Volume 62, pp. 1–66, 2004
[2] K. Beck, *Test Driven Development: By Example*, Addison-Wesley Professional, 2002
[3] A. Perez and R. Abreu, *A Diagnosis-based Approach to Software Comprehension*, in proc. of the 22nd International Conference on Program Comprehension (ICPC 2014), pp. 37–47, ACM, 2014
[4] S. R. Chidamber and C. F. Kemerer, *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, Jun. 1994
[5] E. Arisholm, L. C. Briand and A. Føyen, *Dynamic Coupling Measurement for Object-Oriented Software*, IEEE Transactions on Software Engineering,vol. 30, no. 8, pp. 491–506, August 2004
[6] H. Zhu,P. A. V. Hall and J. H. R. May, *Software Unit Test Coverage and Adequacy*, ACM Computing Surveys, Vol. 29, No. 4, December 1997
[7] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008
[8] M. Fowler, *Test Coverage*, April 2012, http://martinfowler.com/bliki/TestCoverage.html, accessed June 2014
[9] Apache Commons Email website: http://commons.apache.org/proper/commons-email/, accessed April 2014
[10] EclEmma website, http://www.eclemma.org/, accessed June 2014
[11] P. Tahchiev, F. Leme, V. Massol, and G. Gregory, *JUnit in Action, Second Edition*, Manning Publications, 2010
[12] R. E. Jeffries, *You’re NOT gonna need it!*, http://www.xprogramming.com/Practices/PracNotNeed.html, accessed April 2014
[13] W. Li and S. Henry, *Maintenance metrics for the object oriented paradigm*, in proc. of the First International Software Metrics Symposium, pp.52–60, IEEE, 1993
[14] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 1st edition, Addison-Wesley Professional, 2000
[15] A. Tahir and S. G. MacDonell, *A Systematic Mapping Study on Dynamic Metrics and Software Quality*, in proc. of the 28th IEEE International Conference on Software Maintenance (ICSM), pp. 326-335, IEEE, 2012
[16] Á. Mitchell and J. F. Power, *Using object-level run-time metrics to study coupling between objects*, in proceedings of the 2005 ACM Symposium on Applied Computing (SAC ’05), pp. 1456-1462, ACM, 2005
[17] Á. Mitchell and J. F. Power, *A Study of the Influence of Coverage on the Relationship Between Static and Dynamic Coupling Metrics*, Science of Computer Programming, vol. 59, no. 1–2, pp. 4–25, January 2006
[18] R. Geetika and P. Singh, *Dynamic Coupling Metrics for Object Oriented Software Systems- A Survey*, ACM SIGSOFT Software Engineering Notes, Volume 39, Number 2, March 2014
[19] L.C. Briand, S. Morasca and V.R. Basili, *Property-based software engineering measurement*, IEEE Transactions on Software Engineering, vol.22, no.1, pp.68–86, Jan 1996