SPRINT SMEs Project: Research in Software PRocessImprovemeNT Methodologies for Greek Small & Medium sized Software Development EnterpriseS

Work Package 3 (WP3): Design of SPINT SMEs Knowledge Base

Deliverable 3.3 (D3.3): Knowledge Management of Software Processes in Open Source Development Projects

Scientific Coordinator / Project Leader:

Vassilis C. Gerogiannis, PhD, Associate Professor, Dept. of Business Administration, Technological Education Institute of Thessaly, Greece

Authors:

- George Kakarontzas (WP3 Leader), PhD, Applications Professor, Computer Science and Telecommunications Department, Technological Education Institute of Thessaly, Greece
- Ioannis Stamelos, PhD, Associate Professor, Dept. of Informatics, Aristotle University of Thessaloniki, Greece
- Stamatia Bibi, PhD, Research Associate, Dept. of Informatics, Aristotle University of Thessaloniki, Greece
- Leonidas Anthopoulos, PhD, Assistant Professor, Dept. of Business Administration, Technological Education Institute of Thessaly, Greece

May, 2014

Research in SPRINT SMEs project is implemented through the Operational Program "Education and Lifelong Learning" and is co-financed by the European Union (European Social Fund) and Greek national funds in the context of the R&D program ARCHIMEDES III

Table of Contents

List of Figures

List of Tables

1. Introduction

Open source software represents an invaluable resource for software development SMEs. The reason is that it can be reused in order to produce new software products. However it is not easy to understand the source code of Open Source projects because usually they lack documentation regarding design. This lack of *design knowledge* is the single most important impediment in open source software reuse along with the sometimes non-permissive license that prohibits commercial exploitation.

To this end SPRINT-SMEs project contributes in the Knowledge Management of Software Processes in Open Source Development Projects by providing techniques that are based on tests for open source knowledge acquisition and comprehension.

More specifically static analysis (e.g. reengineered UML diagrams) can be useful but very often static analysis is not accurate in the sense that it can be affected by wrong design decisions that distort the true associations between the various modules or that existing associations are not differentiated enough. This report 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 among classes. We demonstrate with an illustrative example of the popular Apache Email open source component that this new metric can provide additional information in reverse engineered class diagrams highlighting missing associations, 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 used among other things, also during open source software project reuse as comprehension aid. Since EBC is based on tests, Open Source software design knowledge becomes easier to obtain since although open source projects often lack detailed documentation they usually have extensive test suites.

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. However they have not been used extensively as design aids or as comprehension aids during program maintenance. In this work we 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.

To explain the basic idea, assume the situation depicted in Figure 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 Figure 1. In addition, instances of classes A and B both call the method m3() on an instance of class C. This is depicted in the UML sequence diagram at the bottom of Figure 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 the same method in called.

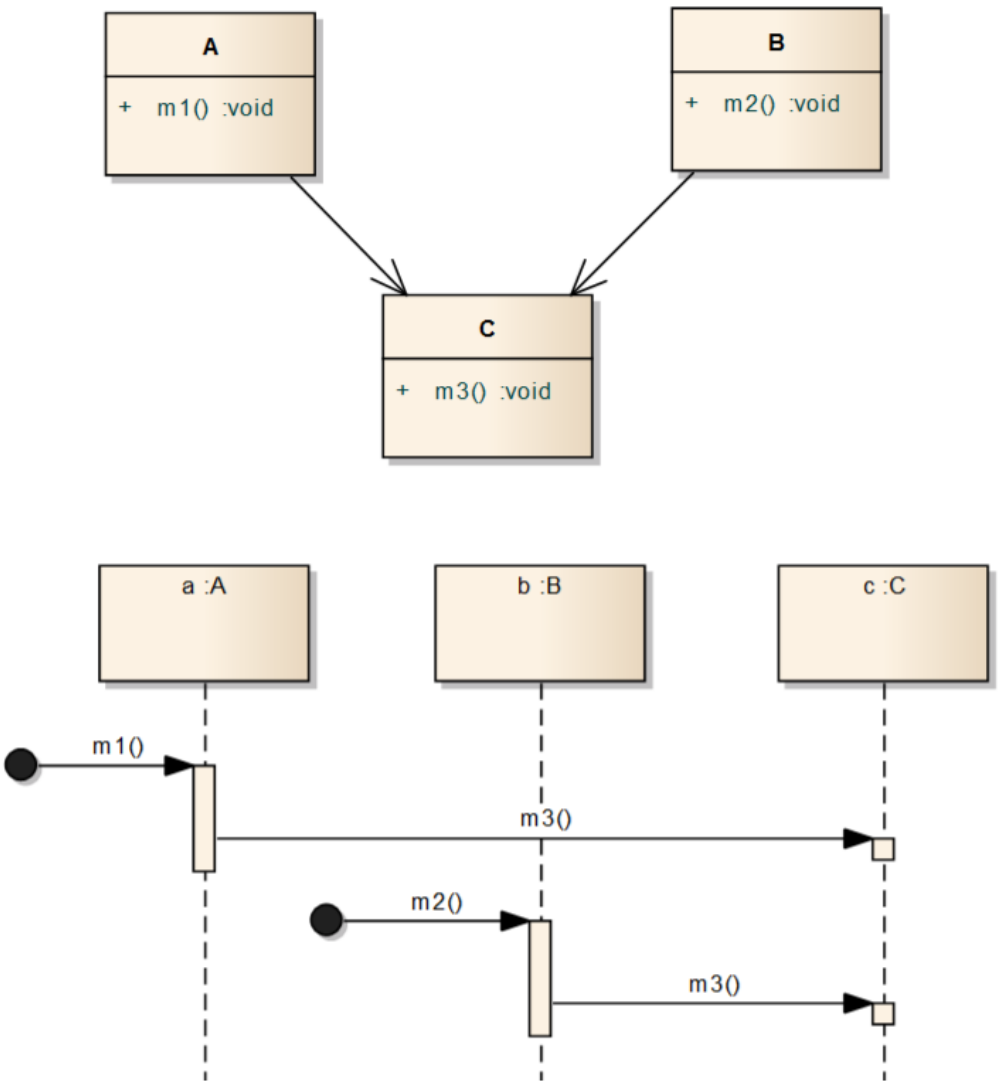

Figure 1: Example class and sequence diagrams

However the situation may be different if we examine the number of statements in m3() executed as a result of the first call and the number of statements in m3() executed as a result of the second call. This information may be very important since m3() may have tens or even hundreds of lines of code. Each different call by different methods represents a distinct context which may cause completely different behaviors in the server object; in our case the instance of class C. In the first case a few lines may be enough, whereas in the second the whole of method m3() may be exercised. Static coupling measures, such as the

popular Coupling Between Objects (CBO) [3] and dynamic measures, such as the suite proposed in [4], although they highlight coupling they do not provide this information. Intuitively, however, this 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.

In this report we explore the idea that statement coverage may provide also an indication of the strength of association between classes or the lack of it. Statement coverage is an adequacy criterion in testing [5], [6] in which "the percentage of the statements exercised by testing is a measurement of the adequacy" [5]. In Section 2 we discuss the idea of Estrangement Between Classes (EBC) based on test coverage and introduce the definition of the EBC metric. In the following Section 3 we present the results obtained using the EBC metric in the popular Open Source Apache Commons Email component. Next in Section 4 we discuss related works and in Section 5 we conclude the report and briefly discuss future research directions.

2. 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 also passes from class B if A uses B or otherwise depends on it. 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 (A, A1 and A2) but during

integration testing control also passes from classes on the right-hand side (B, B1 and B2 respectively). Intuitively we can argue that class A1 is more related to class B1 than class A is related to class B. This is because although we achieve the same coverage in both A and A1 (the shaded area) the coverage in class B1 is more than that in class B. Similarly we can argue that A2 is more related to B2 than A1 is to B1 because the same coverage is achieved in both cases to the right-hand side class (B1 or B2) however this coverage is achieved in the case of B2 with less effort since the coverage in A2 is smaller compared to the coverage in A1.

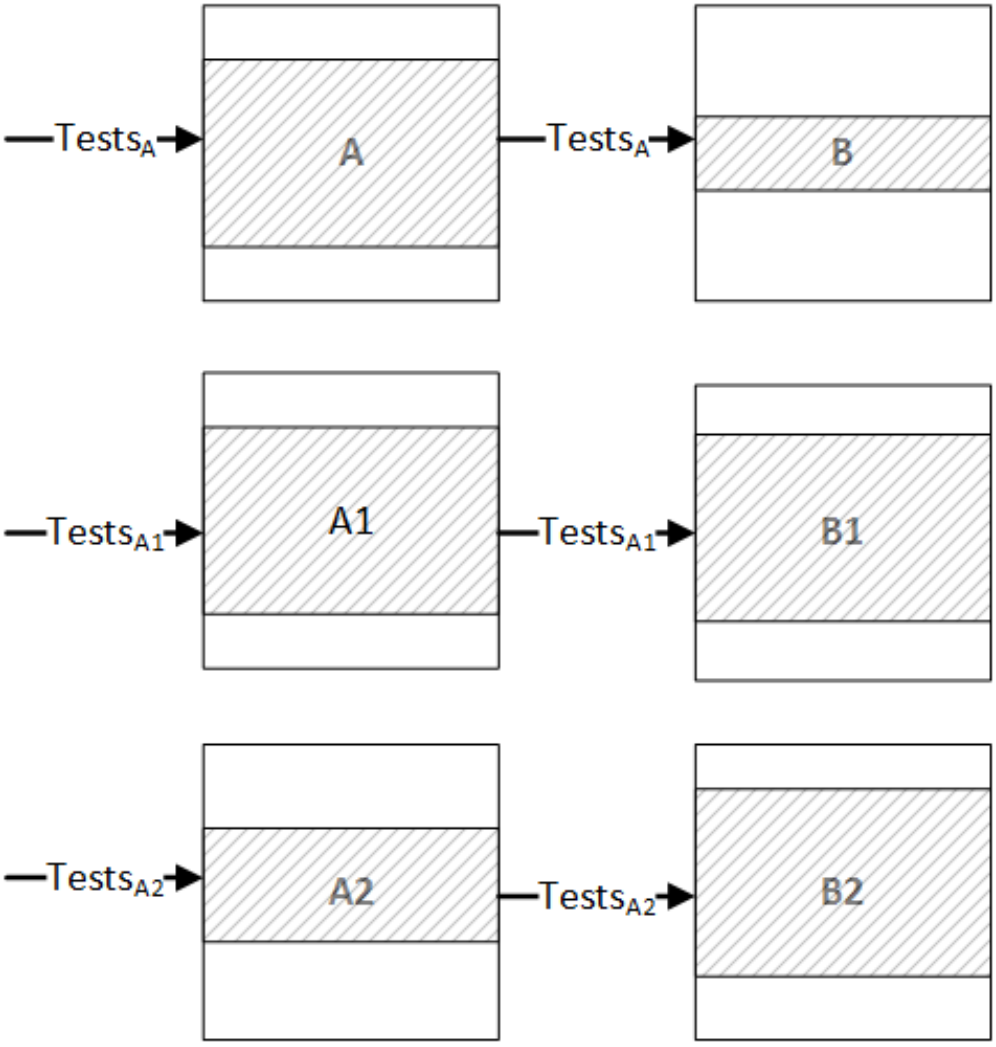

Figure 2: Coverage graphic examples

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 A divided by the coverage of A. The denominator signifies the extent that class A is tested by the same test suite. In the following we will use $C_{TSA}(B)$ to indicate the statement coverage of test suite of class A in class B. Then EBC from class A to class B is defined by Eq. 1:

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

In Equation 1:

1. We subtract the strength of the association as denoted by the fraction from 1 to get the estrangement between classes. If two classes are related less, then this fraction will be smaller and EBC will be larger. Thus EBC takes larger values for less related classes and takes values in the range [0, 1]. Therefore it measures the lack of coupling.

2. In Eq. 1 the bottom case is necessary to avoid negative EBC values when the statement coverage in B surpasses that in A.

3. EBC is not a symmetric since in the general case $EBC(A, B) \neq EBC(B, A)$.

4. Estrangement is relative to the extent that A was tested since B's statement coverage (numerator) is divided by A's test coverage (denominator). 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 metric can only be effective if tests have a significant test coverage. Otherwise the metric cannot provide the desired information. For example if a class is not tested at all, the denominator may be zero which makes EBC undefined. This makes EBC 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 should be used as an indicator if $C_{TSA}(A) \geq 70\%$ although the actual threshold must be determined with empirical studies that we will conduct in our future work.

3. EBC Case Study: The Open Source Apache Commons Email Component

In order to illustrate the use of EBC we used existing unit tests of the Apache Commons Email ver. 1.3.2 [7] which is the current release at the time of this writing 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. 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 attach- ments".

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 second branch of Eq. 1 for the avoidance of negative EBC values. In addition classes *EmailAttachment* and *EmailUtils* also present higher test coverage values than the *HtmlEmail* class with 84% and 82.9% statement coverage respectively. All the results of running the *HtmlEmailTest* and the EBC of each class to the target *HtmlEmail* class is shown in Table 1.

Table 1: EBC BETWEEN HTMLEMAIL AND OTHER CLASSES BASED ON COVERAGE OF HTMLEMAILTEST

Class	Missed	Covered	Coverage	EBC
Default Authenticator	0	13	100.00%	**0.00%**
Email Attachment	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%**
ByteArrayData Source	204	0	0.00%	**100.00%**
EmailConstants	3	0	0.00%	**100.00%**
ImageHtml Email	149	0	0.00%	**100.00%**
SimpleEmail	17	0	0.00%	**100.00%**

According the the results in Table 1 the most estranged classes to the *HTMLEmail* class are *SimpleEmail*, *ImageHtmlEmail*, *EmailConstants* and *ByteArrayDataSource* class. 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* according to the description of the class is a typed data source and not specific to the HTML Email core functionality and furthermore is also deprecated in the tested release. Class *EmailConstants* contains only static fields and no methods and *ImageHtmlEmail* is a subclass of *HTMLEmail* (see Figure 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. 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 is shown in Table 2.

Table 2: EBC BETWEEN SIMPLEEMAIL AND OTHER CLASSES BASED ON COVERAGE OF SIMPLEEMAILTEST

Class	Missed	Covered	Coverage	EBC
Default Authenticator	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%
Email Exception	52	4	7.10%	92.90%
ByteArray DataSource	204	0	0.00%	100.00%
Email Attachment	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 Figure 3.

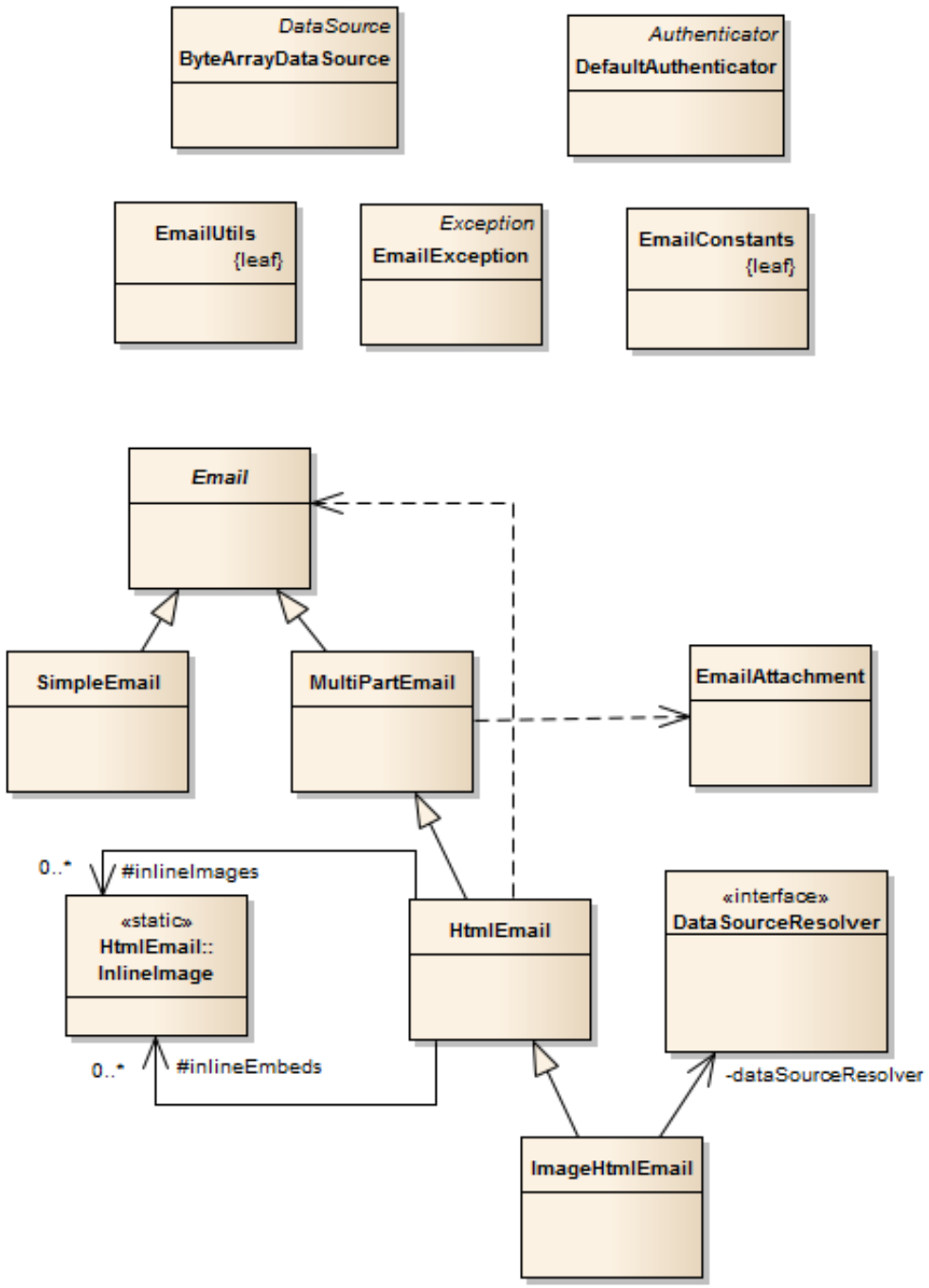

Figure 3: UML Class Diagram of the email component

Notice however 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 3) as well as for *HtmlEmail* 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% and to *MultiPartEmail* by 52.57%. Therefore, although inheritance is a stronger association than (UML) dependency EBC suggests differently. EBC provides therefore evidence for the strength of the associations between classes signifying which are more important than others in terms of code usage.
- 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.
- Some classes are not used at all. For example class *ByteArrayDataSource* is completely estranged to all tested classes in our example (see Table 1, Table 2 and Table 3). 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.

Table 3: 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%

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 and identical signature with the *MutliPartEmail* method:

```
public abstract MultiPartEmail attach(EmailAttachment attachment)
throws EmailException;
```

This 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. 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 Figure 4.

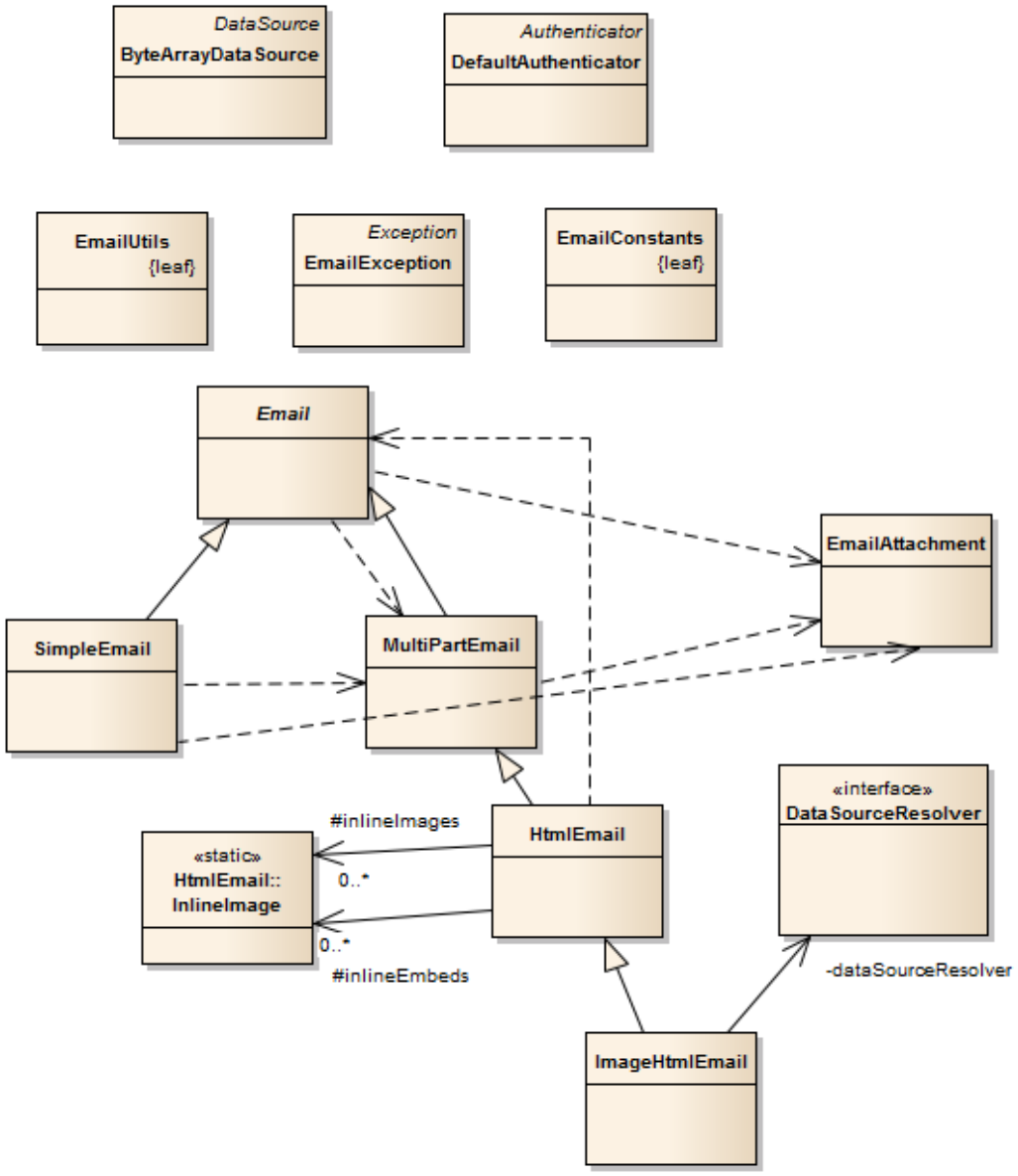

Figure 4: UML Class Diagram of the email component after introducing a wrong design decision

Looking at this 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 refactor the *MockEmailConcrete* class which extends *Email* and is used for the tests, by introducing a null implementation of the *attach* method.

The *HtmlEmailTest* and *SimpleEmailTest* were executed successfully and the following Table 4 and Table 5 contrast the original results with the results obtained from the refactored component.

Table 4: 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 5: 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%

Regarding the *HtmlEmail* test in Table 4 the results are exactly the same as the original results and regarding the *SimpleEmail* test in Table 5 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 drops slightly although their test coverage is the same.

Consequently the tests 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 Figure 4. Notice that 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 the target class coverage is in the denominator).

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" [8].

4. Related Works

The work described here can 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.

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 [3]. 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. [9] 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 [3]. According to [9] 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. [4] 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 client or user side whereas export coupling measures coupling from the server or provider side. The suite of the proposed metrics in [4] 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 [4] since it can provide an assessment of the strength of the association between two classes.

Mitchell and Power [10] 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 [10] observe, is related to the static CBO metric [3] 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 [11] 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 where the six class metrics proposed by Arisholm et al. in [4]. Initially the authors show, using Principal Component Analysis (PCA) that the dynamic metrics proposed in [4] provide additional information than the CBO metric [3] and they are not surrogate metrics compared to CBO. Then they used statistical regression in which each one of the six dynamic metrics where 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 [4] 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. Intuitively this is anticipated since the more code we cover with the tests the best we capture the dynamic behavior with a dynamic coupling metric. In our work we formulate a metric based on test coverage and use this directly as an indicator of the lack of coupling between classes.

Another recent survey on dynamic coupling metrics is provided in [12]. 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. [4]. None of the metrics described in [12] use test coverage as a criterion to decide coupling of objects.

5. Conclusions and Future Research Directions

We proposed Estrangement Between Classes (EBC), a measure based on test coverage, as a tool 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 metric can compare the strength of associations in statically generated class diagrams or highlight missing associations in such diagrams. It 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 in the face of wrong design decisions with an example of the popular Apache Commons components, namely the Email component example. 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, to highlight the correct modular structure for the application based on measurable evidence produced by the test coverage of the developed tests. They can also be used to generate or reinforce the design knowledge of Open Source projects during reuse.

Future steps include the investigation of additional coverage measures such as branch coverage and path coverage and the empirical validation of the proposed metric in a number of projects from available open source repositories. Also a very important issue is the determination of the actual threshold of test coverage required to make the proposed metric useful. Intuitively such a threshold exists and we have suggested here a threshold of 70%. However it will be very useful to determine this value

empirically and also to determine if such threshold is also relevant for other dynamic coupling metrics.

6. References – Bibliography

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. 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
4. 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
5. 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
6. P. Ammann and J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008
7. Apache Commons Email website: http://commons.apache.org/proper/commons-email/, accessed April 2014
8. R. E. Jeffries, Youre NOT gonna need it!, http://www.xprogramming.com/Practices/PracNotNeed.html, accessed April 2014
9. 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

10. A´ . 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
11. A´ . 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
12. 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