A Test-to-Code Traceability Method using .NET Custom Attributes

Azadeh Rafati[1], Sai Peck Lee[1], Reza Meimandi Parizi[2], Sima Zamani[1]
[1]Department of Software Engineering, Faculty of Computer Science & Information Technology, University of Malaya, 50603 Kuala Lumpur, Malaysia
[2]School of Computing and IT, Taylor's University 47500 Subang Jaya, Malaysia
azadeh.rafati@siswa.um.edu.my; saipeck@um.edu.my; rezameimandi.parizi@taylors.edu.my; s.zamani@siswa.um.edu.my

ABSTRACT
According to the fact that unit tests are valuable sources of up-to-date documentation, maintaining traceability links between unit tests and production code can be helpful for software engineers to perceive parts of a system. Traceability information facilitates the testing and debugging of complex software by linking the dependencies between code and tests. Researchers have proposed various approaches in order to recover test-to-code traceability links. However, results from applying these approaches are not satisfactory due to low accuracy and manual effort required by a domain expert to verify recovered links. This paper further identifies the problems of existing test-to-code traceability recovery approaches and proposes a new method, called Embedding test-to-code Traceability links into Unit tests via a Custom Attribute (ETUCA).This method establishes traceability links between unit tests and production code with minimal effort required from unit-test developers. ETUCA introduces a .NET custom attribute which acts as a direct and explicit traceability link to be incorporated into every unit test during the unit-test development process for recording the traceability information. Empirical results indicate that, compared to manual tracing, ETUCA notably reduces the time taken for tracing test units in relation to production code. Moreover, it would be able to recover all traceability links, given test developers correctly embedded custom attributes in the development phase.

CCS Concepts
• **Software and its engineering→Software creation and management→Software verification and validation→Software defect analysis→Software testing and debugging;**

Keywords
Traceability recovery; test-to-code traceability; unit test; empirical studies; software testing; software maintenance phase

1. INTRODUCTION
Traceability is the ability to link and trace different artefacts that helps to increase the quality of software during software development and maintenance processes [19]. In order to ensure that a software project is completed on time successfully, researchers have developed traceability techniques in various fields to trace different software artefacts in both forward and backward directions [23]. Traceability can be applied horizontally across the artefacts in one stage of software life cycle or vertically through different levels of software life cycle. In this way, traceability links are formed among the requirements, source code, unit tests, design and other software artefacts [21]. Unit tests are valuable source of live and up-to-date documentations as they are being modified continuously by corresponding developers to reflect changes in the production code. A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work [10; 16]. Coding and testing are two key activities of software development that are tightly intermingled, especially in incremental and iterative methodologies such as agile software development [20; 22]. The trace links between tests (or test cases) and code artifacts (hereafter called test-to-code traceability) are typically implicitly presented in the project's source code, forcing developers towards overhead code inspection or name matching to identify test cases relevant to a working set of code components. Maintaining traceability links between unit tests and production code can be useful to keep up-to-date documentation as the unit tests are being modified continuously by developers to reflect changes in the production code [16]. However, capturing and creating traceability information as a by-product of development (i.e. performing in parallel with the artefacts) is often said to be tedious [20], and is rarely done to the necessary level of granularity by developers, which can result in unreliable and incomplete recording of the relevant traceability. In this regard, traceability link recovery techniques (as after-the-fact way of establishing traceability information) can aggressively tackle these problems by reducing the associated cost and effort needed to construct and maintain a set of traceability links, and compensating incompleteness of trace information during the development process [13]. However, the existing test-to-code traceability recovery approaches [14; 16; 17; 22] do not recover the traceability links completely and accurately. Further, the recovered links usually do not contain fine-grained information to trace tested methods (limited only to tested classes). Additionally, the recovered links need to be manually verified by a domain expert, thus consuming a great deal of time in large software systems. These observations denote that there is still lack of a method to embed test-to-code traceability links between test cases and the corresponding tested methods which can then be recovered directly with minimal time and effort. Inspired by this, we propose Embedding test-to-code Traceability links into Unit tests via a Custom Attribute (ETUCA). In this method, a .NET custom attribute is proposed to be incorporated into every test case through unit-test development process for recording test-to-code traceability links. The traceability information can then be extracted later to be updated for complete specification of the test-

RACS'15, October 9–12, 2015, Prague, Czech Republic.

DOI: http://dx.doi.org/10.1145/2811411.2811553

to-code links. The retrieved traceability links are not subjected to be verified because they have been incorporated by the corresponding unit-test developers in advance. The remainder of this paper is organized as follows. Section 2 discusses the related works and the existing problems of test-to-code traceability recovery approaches. Section 3 presents the proposed test-to-code traceability method and section 4 describes how it has been implemented. Section 5 presents the empirical assessment, whereas Section 6 reports the achieved results. Section 7 explains the threats to validity, and finally, Section 8 provides the conclusion and highlights the future directions. The proceedings are the records of the conference. ACM hopes to give these conference by-products a single, high-quality appearance. To do this, we ask that authors follow some simple guidelines. In essence, we ask you to make your paper look exactly like this document. The easiest way to do this is simply to download a template from [2], and replace the content with your own material.

2. RELATED WORKS

Achieving artefacts traceability is usually categorized in two ways: By-Product of Development and By Performing After-the-Fact Traceability Mapping. Developers are not usually willing to record traceability information while they are developing the software systems. If traceability information is maintained as a by-product of development which is in the development phase, it would be desirable; But it is also time consuming and rarely done in this stage [20]. As traceability information is usually unavailable in maintenance phase, traceability recovery is performed in an after-the-fact [20] manner. Researchers [16; 17; 22] argued that in most cases of real-world software systems, traceability links are not maintained properly or even documented, and thus, are unavailable during the maintenance phase. Therefore traceability links need to be recovered during the evolution of software systems.

2.1 Traceability Recovery

In the last decade, several approaches had been proposed to recover traceability links between software artefacts. De Lucia et al. [3] classified the approaches for recovering traceability links between various artefacts according to several applied methods to extract links, which are Heuristic based, IR based, data mining based, and machine-learning based. To characterize the heuristic based approaches, Murphy et al. [11] have developed software reflection models. The models are used to allow software developers to compare artefacts by summarizing where one artefact (such as design) is consistent or inconsistent with another artefact (such as the source code). Researchers in [1; 4; 6-9; 27] have applied Information Retrieval (IR) techniques to recover traceability links between different types of artefacts. The rationale behind IR techniques is that most software documentations consist of textual descriptions. Therefore, traceability links could be recovered based on the similarity between the texts contained in the software artefacts. Data mining techniques have been applied to a version repository for recovering traceability links between source code artefacts in [5; 25; 26] based on mining concurrent versions system (CVS) history to detect coupling between modules such as variables, classes, files and functions. Finally, machine-learning methods have been suggested to recover traceability links between regulatory codes and product-level requirements using manual traceability matrices and web-mining techniques in [2]. Test-to-Code Traceability Recovery Approaches Today's integrated development environments provide some degree of support for software developers to navigate between unit tests and tested classes. As an example, Microsoft visual studio 2010 environment provides a wizard to create unit tests for existing classes. In its subsequent versions, a unit-test generator is available as an extension for users. The Eclipse Java environment also provides the same features. Several sources [14-17; 22] have proposed different test-to-code traceability solutions to recover traceability links between unit tests and production code. Sneed [17] introduced two approaches to link test cases and code. The first approach uses statistically name matching to map code functions to requirements model. This approach is unreliable due to missing of some remainin4g links. To solve the problem of this method, the second approach links dynamically test cases and code functions using time stamps between these two artefacts. In the context of dynamic analysis, Quinten et al. [18] investigated how method-level changes in the production code can serve to determine which test cases need to be rerun. In their study, the results of the experiments performed on PMD[1] and CruiseControl[2] revealed that although it is feasible to use the method-level changes to select a subset of the entire test suite which is significantly smaller, however, the selected subset misses some relevant tests [18]. Beside this, dynamic analysis is expensive and time consuming. Rampaei et al. [22] empirically compared a series of automatable traceability approaches, called Naming Convention (NC), Fixture Element Types (FET), Static Call Graph (SCG), Last Call Before Assert (LCBA), Lexical Analysis (LA) and Co-Evolution. The results of their empirical experiments on three open-source Java systems, JPacman, ArgoUML and Mondrian, indicated that although LCBA, LA and FET have high applicability, they have poor results in precision and recall. On the other side, NC showed a good result in terms of accuracy. The best overall result was achieved with a combination of NC, FET and LCBA. Qusef et al. [16] introduced a traceability recovery approach based on Data Flow Analysis (DFA) to enhance LCBA method [22]. This approach works based on the assumption that, if a method call in a unit test affects the result of the last assert statement, the method should belong to the unit under test. To assess the accuracy of DFA, Qusef et al [16] performed a case study on AgilePlanner and Mondrian which are two open-source Java software systems. With this case study, the effectiveness of DFA in the identification of test-to-code traceability links was analysed and DFA was compared with NC and LCBA, in terms of accuracy of the recovered traceability links. The results showed that DFA is more accurate than NC and LCBA. However, DFA failed when there was no real relationship between unit test classes and classes under test that affect the results of the last assert assessment. Qusef et al. [15] also presented an automated approach called SCOTCH (Slicing and Coupling based Test to Code trace Hunter) which is based on the last assert statements analysis, dynamic slicing and conceptual coupling together to identify a set of tested classes in two sequential steps. In the first step, SCOTCH identifies the last assert assessment instance in each execution trace using dynamic slicing. In the second step, it uses a stop-class list (list of classes from standard library types such as java.*, javax.*, org.junit.*) and conceptual coupling to shortlist the candidate types and prune-out unwanted classes. Since only conceptual coupling is used as a filtering strategy and there are more crucial information in the classes, most recently, Qusef et al. [14] introduced an extension to this approach, named SCOTCH+ (Source code and Concept based Test to Code traceability Hunter). This extended version of the approach employs a more sophisticated filtering strategy based on both

[1] http://pmd.sourceforge.net
[2] http://cruisecontrol.sourceforge.net

internal and external textual information, and provides a considerable improvement in accuracy.

2.2 Problems with Existing Test-to-Code Traceability Recovery Approaches

The review of the existing test-to-code traceability recovery approaches provided us with a number of lessons to identify the existing problems in this field. Even SCOTCH+ as the most recent and complete test-to-code traceability recovery approach requires improvements to reach its maturity. In summary, the major problems/issues identified are listed below.

- Test-to-code traceability links are not established completely in all existing traceability recovery approaches.
- Most of the proposed test-to-code traceability approaches recover traceability links between unit tests and the classes under test, and not specifically the methods under test.
- The most important issue with the existing test-to-code traceability approaches is that most of the recovered traceability links need to be verified manually by a domain expert.

Thus, it can be said that there is a lack of reliable test-to-code traceability approach to recover all traceability links between unit tests and production code with minimal human effort for links verification. As the analysis of code is a sophisticated task, we attempt to propose a simple but reliable method for maintaining traceability information with minimal effort for developers. This had motivated us to look into how the proposed method can encourage software developers to participate in maintaining traceability information during the development and testing phases.

3. THE PROPOSED METHOD

In order to improve the aforementioned issues, this study presents a test-to-code traceability method called ETUCA. In this method, traceability links are embedded directly and explicitly into the unit-test source code, so that this valuable traceability information can then be retrieved. We hypothesize that while unit tests are developed, the correct traceability information are incorporated or maintained by unit-test developers. The rationale behind this intuitive idea was inspired by the fact that unit-test developers are well aware of the method or class for which they are writing a unit test. Thus, they can optimally embed traceability links at the same time developing the test projects. This concurrency will help to establish traceability links between unit tests and production code more accurately. Moreover, if traceability information is embedded in the software development phase, no post-verification of recovered links is needed.

3.1 Custom Attribute

Custom Attributes[1] are classes derived from *System.Attribute* class that contains methods to store and retrieve data. *System.Attribute* class is part of Microsoft.NET framework library. *Attributes* provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an *attribute* is associated with a program entity, the *attribute* can be queried at run-time by using a technique called Reflection. *Attributes* add metadata to the program. Metadata is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. Any custom attribute class has to be inherited from "*Attribute*" abstract class to have the base properties. Custom attributes can be added to specify any required additional information. ETUCA takes part in two main phases of software development life cycle: development phase (when unit tests are developed) and maintenance phase (when the system is up and running) as shown in Figure 1.

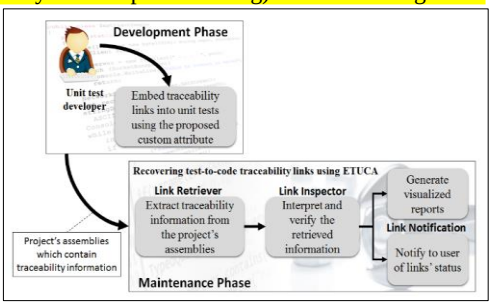

Figure 1. Conceptual view of the proposed method

3.2 Development Phase

To embed traceability links directly and explicitly into the unit-test's source code, a .NET custom attribute, called traceability attribute, is introduced which acts as a direct and explicit traceability link. This traceability attribute is incorporated into every test case through the unit-test development process in order to record traceability information. The traceability attribute can be incorporated easily by unit-test developers at the same time of writing a test case. Therefore, incorporating the attributes is performed at the test development phase and not as by-product of development, given that the incorporation is in the test code and not production code. The whole process of incorporation of the traceability attribute takes an average of 25 seconds for a test case.

To make it easier for unit-test developers to embed traceability information into the traceability attribute, a code snippet has been developed, called ETUCA Snippet, which creates an empty test method with the traceability attribute. Code snippets[2] are ready-made lines of codes which can be quickly inserted into the code editor.

```
[Test]
[TestToCodeLink("Qualify", typeof(SqlLogSpy))]
public void QuotedSchemaName()
{
    Table tbl = new Table();
    tbl.Schema = "`schema`";
    tbl.Name = "name";

    Assert.AreEqual("\"schema_table\"",
        dialect.Qualify("", "\"schema\"", "table"));
}
```

Figure 2. Example of the embedded traceability link in test case for a method

Embedding traceability links into test cases using ETUCA snippet saves the time of incorporating traceability attributes up to 50% which means less than 12 seconds instead of 25 seconds for incorporating a traceability attribute. Figure 2 and Figure 3 illustrate how the traceability attribute, TestToCodeLink, is incorporated into unit tests. Figure 2 demonstrates the code example of a test case from Nhibernate project which is an open source Object-Relational Mapper (ORM) for the .NET framework. In this example, the "*QuotedSchemaName*" test case tests the "*Qualify*" method in the "*SqlLogSpy*" class. This information was incorporated into the traceability attribute, TestToCodeLink, to embed the traceability link. Figure 3 shows a test case also from Nhibernate project that was written to test a class rather than a method, i.e. the "*CanUseParametersWithProjections*" test case, tests the "*AddNumberProjection*" class.

[1] http://msdn.microsoft.com/en-us/library/z0w1kczw.aspx
[2] http://msdn.microsoft.com/en-us/library/z41h7fat.aspx

```
[Test]
[TestToCodeLink(typeof(AddNumberProjection))]
public void CanUseParametersWithProjections()
{
    using (ISession session = sessions.OpenSession())
    {
        long result = session.CreateCriteria(typeof(Student))
            .SetProjection(new AddNumberProjection("id", 15))
            .UniqueResult<long>();
        Assert.AreEqual(42L, result);
    }
}
```

Figure 3. Example of the embedded traceability link in test case for a class

3.3 Maintenance Phase

In the maintenance phase, the traceability information is extracted from the project assemblies (which consist of unit tests and product code assemblies) to specify the test-to-code traceability links via a Link Retriever module. Then, to interpret and verify the retrieved traceability links, a Link Inspector module is used. This module investigates the status of the traceability links to determine whether they are valid. To validate the status of the traceability links which are embedded as traceability attributes, the respective test case of each traceability link is analyzed. The result of the verification step specifies the links' status. The recovered traceability links are not subjected to verification at the end because they had been incorporated by the unit-test developers in advance. The corresponding information of the links' status is logged to be utilized with a Notification module which informs users about the links' status. All the retrieved traceability information from Link Inspector module are presented as visualized reports. Web Reporting application and Traceability Console application are provided to present the traceability information in various formats to be helpful for unit-test developers and project managers. With regards to benefits of the proposed method and a short period of time for incorporating the traceability attributes, unit-test developers can be convinced to apply it.

4. IMPLEMENTATION

This section gives the implementation details of ETUCA, making it usable in practice.

4.1 Environment

ETUCA solution has been developed using Microsoft Visual studio 2013 as the project platform. It is comprised of three main layers, namely Business layer, Service layer and Presentation layer that have been coded using C# programming language. In the Service layer, WCF (Windows Communication Foundation) technology was used. The Presentation layer was implemented using ASP.NET MVC 5 (Model-View Controller) framework which is the Microsoft implementation of MVC design pattern.

4.2 Link Retriever

The link retriever module extracts the traceability information from the project assemblies after each code compilation. To perform this action, Reflection technique is used. This technique allows known data types, the enumeration of data types in a given assembly, and the members of a given class or value type to be inspected at run-time. Link Retriever looks for certain attributes of classes and methods to retrieve test and business classes from the project assemblies. Test methods are recognized with certain attributes like [Test] for NUnit and MbUnit, [TestMethod] for MSTest, [Fact] for xUnit.net and etc. After detecting test methods, Link Retriever module finds all test cases which have been marked with the proposed traceability attribute, TestToCodeLink. In addition, the names of test methods that have not been marked with the proposed traceability attribute will be recovered as well to be listed and reported to the software development team members. Extracted information will be passed to Link Inspector module. Since this step is executed after each compilation, the required traceability information will be extracted from the latest code to ensure the links always remain up-to-date.

4.3 Link Inspector

This module investigates the status of recovered traceability links in order to inform users if the embedded information is incorrect or misleading. To specify the links' status, the stored information in each embedded traceability attribute is used. The embedded information in the traceability attributes are inspected in the body of the test case using static call graph analysis and in the extracted business classes. For each test case, Link Inspector sends the name of related production class embedded in the traceability attribute to Link Retriever in order to get the type information of that production class. The type information contains class members such as properties and methods information. Then, Link Inspector investigates to find an overload method in production code which agrees with the method called in the test case in relation to the number of parameters and types of parameters. The output of this inspection determines the "*valid*" or "*invalid*" status of each link to be utilized by Link Notification.

4.4 Link Notification

This module reports the retrieved traceability information from Link Inspector in three different presentation formats as follows:

- Web Reporting application which demonstrates links' status as a pie chart and represents the traceability information in grid format with search capability.
- A console application which helps unit-test developers to identify invalid links. Traceability Console application is available for unit-test developers to check the traceability information logs in the development environment.

Integration with visual studio output window to show the traceability link information after each code compilation.

5. EMPIRICAL ASSESSMENT

This section describes in detail the performed experimentation for the proposed method following the guidelines by Wohlin et al. [24]. The goal of our evaluation was to analyse the support given by ETUCA, in terms of time spent, with the purpose of evaluating the usefulness of ETUCA with respect to manual tracing. In order to evaluate the usefulness of ETUCA, we have followed the empirical traceability assessment performed by De Lucia et al. [4]. Furthermore, the quality of ETUCA in three different aspects (usability, correctness and reliability) has been assessed from the users' perspective.

5.1 Subjects

The subjects participating in the empirical experiment were 20 software development team members with different roles and job experiences in software development as well as unit testing. The roles of the participants in the company are Software Developer, Software Tester, Team Leader, Scrum Master and Product Owner with up to 31 years of job experience. The subjects worked with ETUCA as unit testers. This group of participants was chosen to investigate the usefulness of the proposed method from the viewpoint of different roles who have different levels of job experience. This variety of the participants with different roles helps to get better insight of ETUCA.

5.2 Objects

The subjects were asked to perform three traceability recovery tasks over three selected software projects: CommonUtility, Log4net[1] and Nhibernate[2]. We chose these projects because the projects exhibit different characteristics. The scales of the selected projects are small, medium and large, respectively, in terms of Cyclomatic Complexity metric and lines of code. CommonUtility contains the utilities classes which are used in a core banking application. This project belongs to the company at which the experiment was executed. Log4net is an open-source Logging framework for Microsoft .NET Framework. The Logging Services project is intended to provide cross-language logging services for the purpose of application debugging and auditing. Nhibernate is an open-source object/relational mapping tool for .NET environments. Table 1 shows the sizes of the selected projects in terms of Cyclomatic Complexity, lines of code, classes, test classes and test cases. Cyclomatic Complexity[3] measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program. This metric for the selected projects was measured by Visual Studio to represent the project size (and not the maintainability factor). As there is no limitation in the proposed solution, we did not filter any unit tests and the number of all unit tests was reported.

Table 1. Characteristics of the experimented projects

Projects	Characteristics					
					Unit Tests	
	Size	*Cyclomatic Complexity*	*Lines of Code*	*Number of Classes*	*Number of Test Classes*	*Number of Test Cases*
Common Utility	Small	222	457	9	9	70
Log4net	Medium	4,023	7,484	235	29	156
Nhibernate	Large	39,463	78,645	2576	925	3917

5.3 Treatments

The experiment was performed at the participants' workplace and three traceability recovery tasks that the participants had been asked to perform, included:

T1 seeking the test cases associated with two selected methods in the CommonUtility project;

T2 seeking the test cases associated with two selected methods in the Log4net project;

T3 seeking the test cases associated with two selected methods in the Nhibernate project.

The experiment was organized through different sessions. The subjects applied two different methods to perform the traceability recovery tasks. In particular, in the first method, the subjects manually performed the tasks, while in the second one, they performed the task using ETUCA Web Reporting application as automatic tracing[4]. This experiment allows us to compare the time spent by the participants using two methods (manual and automatic using ETUCA) applied in each experiment to find the correct links. This comparison provides a baseline for analysing the usefulness of ETUCA. After the experiment, the subjects were asked to fill up an online survey questionnaire which enables us to assess the quality of ETUCA from three aspects; usability, correctness and reliability. As mentioned in Section 3, the assumption underlying the proposed method is that the traceability information is embedded into the traceability attribute. Thus, to perform the experiment, the three selected projects should have the proposed traceability attributes. Therefore, test-to-code traceability links can be established. As two of the three chosen projects are open-source, the time was limited, and also the scale of the projects was relatively large, a subset of test classes and test cases were randomly selected and few domain experts incorporated the traceability attributes into the selected unit tests. The chosen projects, namely CommonUtility, Log4net and Nhibernate, consist of 70, 156, 3917 test cases, respectively and the traceability attributes were incorporated into only 70, 36 and 94 test cases of these projects, correspondingly. CommonUtility project belongs to the company at which the experiment was executed and the traceability attributes were embedded into the test cases during the development phase.

5.4 Hypotheses Formulation

The goal of evaluating the proposed method was to analyse:

1. How the use of ETUCA affects the tracing performance of subjects during traceability link identification in terms of time spent.
2. How the quality of ETUCA can satisfy the software development team members in terms of usability, correctness, and reliability.

Thus, the null hypotheses are formulated as follows:

H_{0t} the use of ETUCA does not significantly reduce the time spent by the software development team members to trace the links over the manual effort.

H_{0q} the usability, correctness and reliability of ETUCA do not satisfy the software development team members in terms of software quality factors.

If the null hypotheses can be rejected, the corresponding alternative hypotheses will be accepted. Consequently, the alternative hypotheses are:

H_{at} the use of ETUCA significantly reduces the time spent by the software development team members to trace the links over the manual effort.

H_{aq} the usability, correctness and reliability of ETUCA satisfy the software development team members in terms of software quality factors.

5.5 Experimental Design and Process

The experiment was carried out individually for each participant. The participants were asked to identify the corresponding test cases for the given methods, A-F, chosen by the authors with manual and automatic tracing using ETUCA. In order to calculate the time spent for finding traceability links during the experiment, a simple application, called EvaluationTimer, was developed to record the time spent. The participants should discover the corresponding test cases for method A, method B and method C in CommonUtility, Log4net and Nhibernate projects, respectively, with manual tracing method. Then, the same scenario goes for method D, method E and method F, with automatic tracing using ETUCA Web Reporting Application. Meanwhile, the time spent by unit testers for these six cases was recorded by EvaluationTimer application. The time was measured in second. Before the experiment, a description of the goals of the traceability recovery was presented to each subject with detailed instructions on the tasks to be performed. The experimental design allows us to assess the support given by ETUCA in comparison with manual tracing. At the end of each

[1] http://logging.apache.org/log4net/
[2] http://nhforge.org/
[3] http://msdn.microsoft.com/en-us/library/bb385914.aspx
[4] The automatic tracing refers to retrieving the traceability links and not the incorporation of custom attribute.

experiment, the subjects filled up an online survey questionnaire to investigate three aspects of quality of ETUCA including usability, correctness and reliability, as shown in Table 2. The questionnaire is composed of questions with closed answering according to the Likert scale [12], from strongly agree (5) to strongly disagree (1). To statistically evaluate the level of agreement for the quality assessment, the One Sample Wilcoxon is conducted. Q1, Q4, Q7 and Q10 evaluate the Reliability of software quality factors. Q2, Q3, Q8 and Q11 are related to Usability factor, and the other remaining questions assess the software in terms of Correctness.

Table 2. Experiment Questionnaire

Id	*Questions*
Q_1	ETUCA establishes test-to-code traceability links accurately
Q_2	ETUCA is easy to use
Q_3	ETUCA is an user-friendly recovery system
Q_4	ETUCA is a reliable recovery system
Q_5	ETUCA reduces time for finding test-to-code traceability links
Q_6	ETUCA is helpful for facilitating maintenance
Q_7	The traceability information provided by ETUCA was enough to find the links
Q_8	The traceability information provided by ETUCA was useful and meaningful
Q_9	ETUCA saves substantial time in finding links
Q_{10}	I found ETUCA as an efficient traceability recovery system
Q_{11}	I understood how to use ETUCA easily

6. RESULTS AND DISCUSSION

This section reports the results obtained in the experiment, from performing the trace link seeking in the manual and automatic methods. Then, the results of the survey that assess some aspects of quality of automatic tracing are presented in the next part. Finally, the discussion on the obtained results is given in the last part.

6.1 Execution Time

In order to test the time hypothesis (H_{0t}) we analysed the effect of two methods (manual and automatic tracing) on the time metric in the experiments. While the participants performed the experiments, the spent time was recorded to investigate the usefulness of the automatic tracing using ETUCA. One of the main goal of this particular evaluation was to show the short time spent on performing ETUCA to encourage the developer to use it instead of relying on the common traceability methods that execute in the maintenance phase. Figure 4 visualizes the comparison of time spent on the execution experiment on the object projects using manual and automatic tracing. As shown in this figure, the spent time on automatic method (using ETUCA) is remarkably lower than the spent time in manual tracing. The average of the execution time in CommonUtility project for the manual and automatic tracing methods are around 67 and 36 seconds, respectively. The *average* of time difference between these two tracing methods is around 33 seconds. For the Log4net project, these values are 198, 51 and 149 seconds, respectively. For the Nhibernate project, the average of the execution time in manual tracing is 326 seconds while in automatic tracing using ETUCA is 62 seconds. The *average* of execution time difference between manual and automatic tracing in Nhibernate that is a large-scale project is approximately 264 seconds which reveals a remarkable difference.

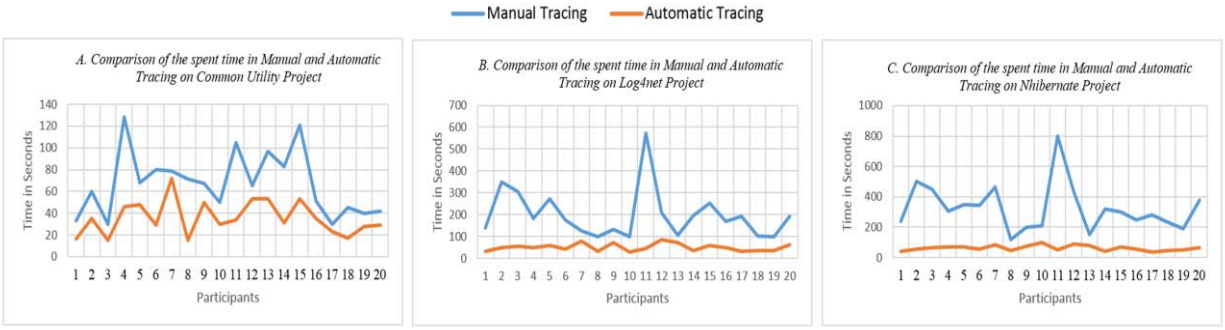

Figure 4. Comparison of the spent time in Manual and Automatic Tracing on the object projects

6.2 Statistical Execution Time Results

In order to assess the normality of experimental data as part of statistical test requirement, the paired T-test was conducted with the significance level of $\alpha = 0.05$. Furthermore, the effect size of Cohen's d for the difference between the methods was reported to complement the inferential statistics.

Table 3. Paired T-Test Results

Projects	*P-Value*	*Effect Size*
Common Utility	*0.9 e-5*	1.362
Log4net	*1.3 e-5*	1.818
Nhibernate	*0*	4.174

Also, Skewness and Kurtosis of the results are between -2 and 2 that confirm the normality of the results. Table 3 reports the results of the Paired T-Test results on the spent time in the manual and automatic tracing process (p-value) and the effect size of the difference between these two processes. The p-value on all projects indicates the significance of the differences and the effect sizes suggest a large practical significance level. Therefore, the first hypothesis, H_{0t} was rejected and H_{at} was accepted, accordingly.

6.3 Survey Results

In addition to investigating the execution time of ETUCA, we analysed the feedback of the participants which were collected from the survey. As mentioned in Section 5.5, the survey mainly aimed at assessing the quality of the automatic method as well as better understanding of the experimental results. The average of the responses for each question was compared with the Neutral level of satisfaction by conducting the One Sample Wilcoxon to evaluate the significance of the difference between the average and the value of Neutral. Table 4 reports the results of Wilcoxon test. According to the results achieved, the participants were *Strongly agreed* on Q_2, Q_5, Q_9 and Q_{11}, as the *mean* value is around 4.6. For the remaining questions, the responses were *Agreed*, with approximately, a *mean* value of 4.2. **Figure 5** summarizes the agreement level of the subjects who assessed the quality of ETUCA in three aspects (when using ETUCA web reporting application). These results were obtained from the average of the responses for the associated questions with each aspect of the

quality. The mean average of the results for the usability, correctness and reliability aspects were around 4.4, 4.5 and 4.2 that indicates good quality of ETUCA from the participants' perspective.

Table 4. One Sample Wilcoxon Test Results

Id	*Mean*	*Median*	*P-value*
Q_1	4.45	4.00	0.000
Q_2	4.50	4.50	0.000
Q_3	4.20	4.00	0.000
Q_4	3.85	4.00	0.000
Q_5	4.65	5.00	0.000
Q_6	4.20	4.00	0.000
Q_7	4.30	4.00	0.000
Q_8	4.30	4.00	0.000
Q_9	4.60	5.00	0.000
Q_{10}	4.20	4.00	0.000
Q_{11}	4.60	5.00	0.000

The analysis of the results with one sample Wilcoxon also revealed a significant difference between the average of the responses and the Neutral level. Hence, the corresponding hypothesis, H_{0q}, was rejected and H_{aq} was accepted.

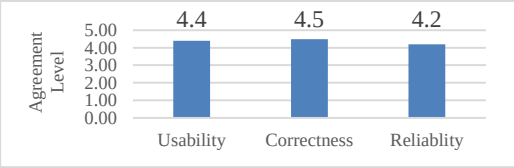

Figure 5. The agreement level in assessing the quality of ETUCA

6.4 Discussion

The empirical experiment was performed to evaluate the performance of the proposed method. The analysis of the descriptive and statistical results for the automatic tracing method compared to the manual tracing showed that ETUCA notably reduces the spent time for tracing the test units into the production code, as shown in Figure 6. That is to say the spent time for automatic tracing of the unit tests in CommonUtility project which is a small project, is not significantly different from the spent time for this process in the other two systems, Log4net and Nhibernate. In other words, as shown in Figure 6, the spent time in the automatic tracing process from the small projects to the large projects increases gradually and the difference is not considerable. While the time spent to find the correct links in the manual tracing process is significantly increased from the small-scale project to the large-scale ones. In addition, the slope of the line that connects the executing time of the automatic tracing on the object systems is not very steep, but the slope of the line in the manual tracing is quite sharp. These findings could be encouraging to recommend ETUCA method for tracing test units to the production code. For assessing the quality of ETUCA, we analysed the survey questionnaire that evaluates the level of agreement of participants in using ETUCA. In this survey, the quality of ETUCA was investigated from three aspects of software quality factors including usability, correctness and reliability. The participants' responses to the respective questions to each one of these aspects indicate the *Strongly Agreed* and *Agreed* levels of agreement. These results revealed that ETUCA has the high quality from the user perspective. ETUCA maintains traceability links information directly and explicitly in the unit-tests' source code. Therefore, traceability links are always available to be utilized without any dependency to external resources such as files or relational databases. ETUCA supports a wide range of unit testing frameworks and has been designed to be easily extensible to support more demanded frameworks.

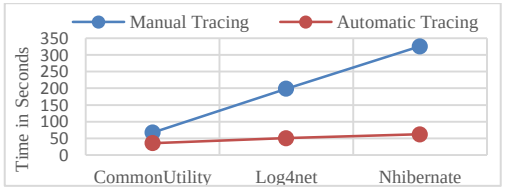

Figure 6. The comparison of time difference on the object systems

It establishes traceability links between production code and unit tests which could either be a class or method. Therefore, it is possible to trace the links from test cases, or test classes to production code classes or methods, and vice versa. This helps unit-test developers to easily seek related unit tests and test cases for every single class or method in production code and aid them to thoroughly test the modified modules after changing the code to assure the correctness of the alterations being made. Additionally, as the traceability links are established either after project compilation or using ETUCA web reporting application, the links are always up-to-date with latest modification. In the technical aspect, .NET custom attributes have been used to embed the traceability information; however, there is no limitation to implement ETUCA in other environments. For instance, Java annotations can be used to implement ETUCA in Java platform.

7. THREATS TO VALIDITY

This section presents the threats to the validity of this study. Firstly, as an internal validity of this study, it was assumed that the unit-test developers who write unit tests embed the corresponding traceability information correctly at the same time with developing the unit tests. With regard to this assumption, we can infer that ETUCA establishes all the traceability links. The factor of human errors in inserting invalid links or missing some is not the issue to be dealt with in this study. However, if the embedded traceability information is incorrect or misleading, ETUCA will have to inform the unit-test developers. To mitigate this threat, ETUCA has an intrinsic support to find the test cases without embedded traceability links, look for the embedded links in the body of the test cases, identify the invalid links (via Link Inspector), and notify the unit-test developer to be aware of them(via Link Notification). Secondly, as the external validity, the reliability of ETUCA has been measured through the survey, but survey results are not sufficient to indicate the system reliability. System reliability requires more variables to be measured. However, based on user experience and the results extracted from the survey, it can be concluded that users agreed with the idea that ETUCA is a reliable system. The final external validity is about generalizing the obtained results from the object systems to other datasets. But it has to be said that we selected different project sizes of from the open-source and commercial projects to avoid specifying the results. Despite this, it cannot be claimed that these results would be similar for all other open-source or commercial software projects.

8. CONCLUSION AND FUTURE WORK

This paper presented a practical test-to-code traceability method, ETUCA, which introduces a .NET custom attribute that acts as a direct and explicit traceability link to be incorporated into every unit test during the unit-test process for recording the traceability information. The traceability information can then be retrieved later to be updated for complete specification of the test-to-code traceability links. The usefulness of the proposed method was empirically evaluated. The analysis of the descriptive and

statistical results for the automatic tracing using ETUCA compared to the manual tracing confirmed that this method notably reduces the time taken for tracing the test units into the production code. Hence, the resulting links found by ETUCA were reasonably good for test-to-code traceability, thus it could be worth using as a useful and efficient tool. Furthermore, to assess the quality of ETUCA, we analysed the survey questionnaire that evaluates the level of agreement of the participants while utilizing this method. In this survey, the quality of ETUCA was investigated from the three aspects of software quality factors including usability, correctness and reliability. The results revealed that ETUCA has the high quality from the users' perspective. There are various directions that can extend ETUCA. The first one is to extend the custom attribute introduced by ETUCA to establish traceability links between more software artefacts such as requirements, design, diagrams, and so forth. The second one is to facilitate working of ETUCA by integrating it with an IDE and utilizing built-in capabilities provided by the IDE.

9. ACKNOWLEDGMENT

This work is supported by High Impact Research Grant with reference UM.C/625/1/HIR/MOHE/FCSIT/13.

10. REFERENCES

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering 28*, 10, 970-983.

[2] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J., 2010. A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* ACM, Cape Town, 155-164.

[3] De Lucia, A., Fasano, F., and Oliveto, R., 2008. Traceability management for impact analysis. In *Frontiers of Software Maintenance* IEEE, Beijing, 21-30.

[4] De Lucia, A., Oliveto, R., and Tortora, G., 2009. Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Engineering 14*, 1, 57-92.

[5] Gall, H., Hajek, K., and Jazayeri, M., 1998. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance* IEEE, Bethesda, 190-198.

[6] Hayes, J.H., Dekhtyar, A., and Osborne, J., 2003. Improving requirements tracing via information retrieval. In *Proceedings of 11th IEEE International Conference on Requirements Engineering* IEEE, Monterey Bay, 138-147.

[7] Hayes, J.H., Dekhtyar, A., and Sundaram, S.K., 2006. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering 32*, 1, 4-19.

[8] Lucia, A.D., Fasano, F., Oliveto, R., and Tortora, G., 2007. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM) 16*, 4, 13.

[9] Marcus, A. and Maletic, J.I., 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering* IEEE, Portland, 125-135.

[10] Martin, R.C. and Melnik, G., 2008. Tests and requirements, requirements and tests: A möbius strip. *IEEE Software 25*, 1, 54-59.

[11] Murphy, G.C., Notkin, D., and Sullivan, K.J., 2001. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering 27*, 4, 364-380.

[12] Oppenheim, A.N., 1992. *Questionnaire design, interviewing and attitude measurement*. Continuum.

[13] Parizi, R.M., Lee, S.P., and Dabbagh, M., 2014. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability 63*, 4, 913-926.

[14] Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., and Binkley, D., 2014. Recovering Test-To-Code Traceability Using Slicing and Textual Analysis. *Journal of Systems and Software 88*, 147-168.

[15] Qusef, A., Bavota, G., Oliveto, R., Lucia, A.D., and Binkley, D., 2012. Evaluating test-to-code traceability recovery methods through controlled experiments. *Journal of Software: Evolution and Process 25*, 11, 1167–1191.

[16] Qusef, A., Oliveto, R., and De Lucia, A., 2010. Recovering traceability links between unit tests and classes under test: An improved method. In *Proceedings of the IEEE International Conference on Software Maintenance* (Timisoara, 12-18 September 2010), IEEE, Timisoara, 1-10.

[17] Sneed, H.M., 2004. Reverse engineering of test cases for selective regression testing. In *Proceedings of the 18th European Conference on Software Maintenance and Reengineering* IEEE, Tampere, 69-74.

[18] Soetens, Q.D., Demeyer, S., and Zaidman, A., 2013. Change-based test selection in the presence of developer tests. In *17th European Conference on Software Maintenance and Reengineering (CSMR)* IEEE, Genova, 101-110.

[19] Spanoudakis, G. and Zisman, A., 2005. Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering 3*, 395-428.

[20] Sundaram, S.K., Hayes, J.H., Dekhtyar, A., and Holbrook, E.A., 2010. Assessing traceability of software engineering artifacts. *Requirements engineering 15*, 3, 313-335.

[21] Tsuchiya, R., Kato, T., Washizaki, H., Kawakami, M., Fukazawa, Y., and Yoshimura, K., 2013. Recovering traceability links between requirements and source code in the same series of software products. In *Proceedings of the 17th International Software Product Line Conference* ACM, Tokyo, 121-130.

[22] Van Rompaey, B. and Demeyer, S., 2009. Establishing traceability links between unit test cases and units under test. In *13th European Conference on Software Maintenance and Reengineering (CSMR)* IEEE, Kaiserslautern, 209-218.

[23] Winkler, S. and Pilgrim, J., 2010. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling (SoSyM) 9*, 4, 529-565.

[24] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., and Wessln, A., 2012. *Experimentation in software engineering*. Springer Publishing Company, Incorporated.

[25] Ying, A.T., Murphy, G.C., Ng, R., and Chu-Carroll, M.C., 2004. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering 30*, 9, 574-586.

[26] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering 31*, 6, 429-445.

[27] Zou, X., Settimi, R., and Cleland-Huang, J., 2010. Improving automated requirements trace retrieval: a study of term-based enhancement methods. *Empirical Software Engineering 15*, 2, 119-146.